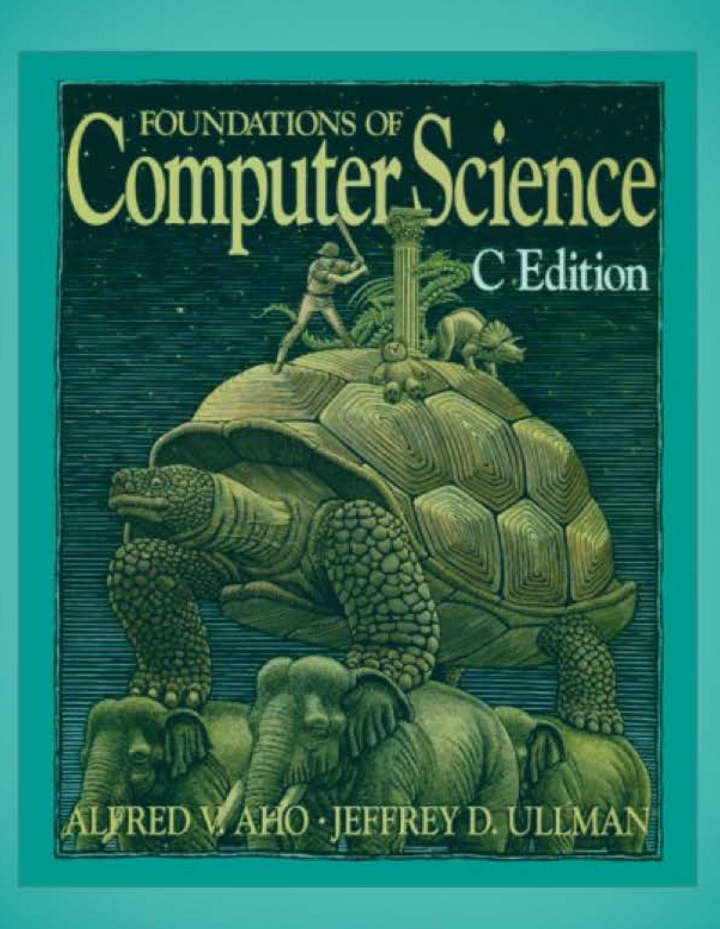
图灵电子书

计算机科学的基础



Al Aho Jeff Ullman

我们相信，在1992年，本书适合用来介绍计算机科学理论，今天仍是。

TURING 图灵计算机科学丛书

Foundations of Computer Science: C Edition

计算机科学的基础

[美]　Alfred V. Aho　Jeffrey D. Ullman　著

傅尔也　译

人民邮电出版社

北　京

图书在版编目（CIP）数据

内 容 提 要

本书以传统计算机科学课程的方式，将数据结构方面的初级课程与离散数学课程结合在一起，全面阐述了计算机科学的理论基础。本书用算法、数据抽象等核心思想贯穿各个主题，从抽象概念的机械化到各种数据模型的建立，很好地兼顾了学科广度和主题深度，帮助读者培养计算机领域的大局观，学习真正的计算机科学知识。

本书适合计算机专业的学生及具有计算机应用技术基础理论知识的读者阅读。

◆ 著　[美] Alfred V.Aho　Jeffrey D.Ullman
译　傅尔也
责任编辑　李　瑛
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：787×1092　1/16
印张：
字数：　千字　2013年12月第1版
印数：1－　000册　2013年12月北京第1次印刷

著作权合同登记号　图字：　号

定价：　.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

广告经营许可证：京崇工商广字第0021号

序

编写本书的动机源于我们对进一步革新计算机科学核心课程的需求。作为对讨论了计算科学入门课程的“Denning 报告”（Denning、P. J.、D. E. Comer、D. Gries、M. C. Mulder A. Tucker、J. Turner 和 P. R. Young，“Computing as a Discipline”，*Comm. ACM* **32**:1，9-23 页，1989 年 1 月）的回应，全美的很多学校都修订了他们的课程。这篇报告引起人们关注作为相关学科所有本科课程之基础的三种工作方法或流程——理论、抽象和设计。最近，ACM/IEEE-CS Joint Curriculum Task Force 的 *Computing Curricula 1991* 报告呼应了“Denning 报告”，特别是确定了作为计算科学之本的那些反复出现的关键概念：概念上和形式上的模型、效率，以及抽象的层次。这两份报告的主题总结了我们试图在本书中提供给学生们的内容。

本书由斯坦福大学一门课程的讲义发展而来，该课程名叫“CS109：计算机科学导论”（CS109：Introduction to Computer Science），它是一门两学季课程，有很多目标，第一个目标是为计算机科学专业初学者的进一步学习打下坚实的基础。不过，计算科学在大量的理工学科中变得越来越重要。因此，第二个目标是为那些不会在计算机科学领域进一步深造的学生提供一些该领域的概念性工具。最后，影响更加广泛的目标是让所有学生了解程序设计概念，并建立扎实的计算机科学知识基础。

本书第一版于 1992 年问世，是基于 Pascal 语言的。当时之所以选择 Pascal 作为示例程序的语言，是因为计算机科学科目的 Advanced Placement[①]考试使用了 Pascal 语言，而且很多大学的程序设计导论课程也是使用 Pascal 语言。我们欣喜地看到，自从 1992 年起，C 语言已渐趋成为主流入门程序设计语言，因此本书这一版的示例程序都用 C 语言写成。本书强调抽象和封装的重要性，这应该能为读者学习涉及使用 C++的面向对象技术的后续课程提供良好的基础。

与此同时，我们决定对本书的内容进行两大改进。首先，虽然对机器的体系结构有所了解有利于激发对度量运行时间的兴趣，但我们发现几乎所有的课程体系都将体系结构单独作为一门课程，所以有关这一主题的章节在这里并不实用。其次，很多计算理论的入门课程会强调组合和概率，所以我们决定增加这方面的内容，并将其单独作为一章。

本书涵盖的主题通常会出现在离散数学课程以及大二计算机科学的数据结构课程中。我们有意从计算机用户的实际需要着眼，选择了数学方面的基础知识，而不是从数学家的角度去选择，并尝试把数学基础知识与计算科学有效地结合起来。因此，我们希望为学习计算机科学的人提供一种比学习程序设计课程、离散数学课程或计算机科学附属学科课程更佳的感觉。相信随着时间的推移，科学家和工程师都将学习与斯坦福大学这门课程类似的基础课程。这样的计算机科学课程也应该像微积分和物理学的相关课程那样成为标准课程。

---

① 简称 AP，指美国高中开设的具有大学水平的课程，即大学预修课程。AP 考试的成绩可折抵大学学分，并成为美国大学的重要录取依据。——编者注

阅读前提

从大一新生到研究生都可能选修基于本书的课程。这里我们假设这些学生都有着扎实的程序设计基础，熟悉本版中用到的 ANSI C 程序设计语言。特别要说的是，我们还希望学生们了解 C 语言中的结构，诸如递归函数、结构体、指针，以及与指针和结构体有关的运算符，如点运算符、->和&。

计算机科学基础课程相关建议

本书以传统计算机科学课程的方式，将数据结构方面的初级课程（也就是 CS2 课程）与离散数学课程结合在一起。我们相信，出于如下两个原因，这些主题的整合是十分必要的。

(1) 把数学与计算科学更加紧密地联系起来，有助于激发对数学的兴趣。

(2) 计算科学与数学是相辅相成的。这样的例子包括，第 2 章中递归程序设计与数学归纳法之间的关系，第 14 章，逻辑学中自由/约束变量的区别与程序设计语言中变量范围之间的关系。此外，与启发性程序设计作业有关的建议纵贯全书。

本书的使用方法有很多种。

两学季或两学期的课程

斯坦福大学的 CS109A-B 系列课程就是典型的两学季课程，不过它们的安排都相当紧，各自要在 10 周时间内完成 40 个课时的教学。这两门课程完整涵盖了本书，其中前 7 章是在 CS109A 中介绍的，而第 8 至 14 章是在 CS109B 中介绍的。

一学期的CS2 类课程

本书也可以用于一学期课程，内容与 CS2 课程的主题类似。本书中的内容确实太多，一个学期自然讲不完，因此我们建议把精力放在以下这些内容上。

(1) 递归算法与递归程序：2.7 节和 2.8 节。

(2) 大 O 分析和程序的运行时间：第 3 章，除了 3.11 节求解递推关系的内容。

(3) 树：5.2 节～5.10 节。

(4) 表：第 6 章。有人可能希望按照更为传统的方式，在介绍树之前先介绍表。我们在这里把树视作更为基础的概念，不过这样调换次序存在一个小问题，就是第 6 章讨论的“词典”抽象数据类型（以及插入、删除和查找操作），在 5.7 节中就作为与二叉查找树相关的概念介绍了。

(5) 集合与关系：7.2 节～7.9 节以及 8.2 节～8.6 节，强调了表示集合和关系的数据结构。

(6) 图算法：9.2 节～9.9 节。

一学期的离散数学课程

对着重于数学基础的一学期课程而言，教员可以选择介绍以下内容。

(1) 数学归纳法和递归程序：第 2 章。

(2) 大 O 分析、运行时间和递推关系：3.4 节～3.11 节。

(3) 组合学：4.2 节～4.8 节。

(4) 离散概率：4.9 节～4.13 节。

(5) 树的数学方面：5.2 节～5.6 节。

(6) 集合的数学方面：7.2 节、7.3 节、7.7 节、7.10 节和 7.11 节。

(7) 关系代数：8.2 节、8.7 节和 8.9 节。

(8) 图算法与图论：第 9 章。

(9) 自动机和正则表达式：第 10 章。

(10) 上下文无关文法：11.2 节～11.4 节。

(11) 命题逻辑和谓词逻辑：第 12 章，第 14 章。

本书特色

为了帮助学生学习这些知识，我们还采取了以下辅助措施。

(1) 每章开头都有内容简介，最后都有小结，用来突出本章要点。

(2) 除了在节标题或小节标题中提到过的概念和定义外，一些重要的概念和定义用楷体突出显示。

(3) 附注栏内容与正文分隔开，这些附注短文有以下用途。

- ❑ 有一些是对正文的详细阐述，或是介绍程序或算法设计的微妙之处。
- ❑ 其他一些是对正文中要点的总结或强调。这类短文包括了对某几类重要证明（比如各种形式的归纳证明）的概述。
- ❑ 少量用于举例说明一些谬误，而且我们希望将其与正文分开可以消除可能出现的误解。
- ❑ 少量非常简要地介绍了像不可判定性或计算机发展史这种要花上一整节来介绍的重要主题。

(4) 几乎每节都有习题，在全书分布着逾 1000 道习题。其中大概有 30%标记了一个星号，表示这些习题比那些不带星号的要多费一番思量。还有约 10%的习题标记了两个星号，它们是最具挑战性的。

(5) 每一章最后还有参考文献。我们不求面面俱到，只不过推荐一些让读者能了解与该章主题有关的高阶教科书，以及那些最具历史意义的相关论文。

封面简介

使用象征图书内容的漫画或图片作为封面是计算机科学教科书的传统。本书用龟背来表示计算机科学的世界，也还有其他很多象征符号代表着那些更高阶计算机科学教科书，本书内容正是为这些书打基础的，这些符号有下面这些。

泰迪熊。R. Sethi，*Programming Languages: Concepts and Constructs*，Addison-Wesley，Reading，Mass.，1989。

棒球运动员。J. D. Ullman，*Principles of Database and Knowledge-Base Systems*，Computer Science Press，New York，1988。

圆柱。J. L. Hennessy 和 D. A. Patterson，*Computer Architecture: a Quantitative Approach*，Morgan-Kaufmann，San Mateo，Calif.，1990。

龙。A. V. Aho、R. Sethi 和 J. D. Ullman，*Compiler Design: Principles, Techniques, and Tools*，Addison-Wesley，Reading，Mass.，1986。

三角龙。J. L. Peterson 和 A. Silberschatz，*Operating Systems Concepts*，第二版，Addison-Wesley，Reading，Mass.，1985。

致谢

很多同事与学生阅读了本书，提出了很多对改进本书表述很有价值的建议，在此我们深表感谢。还要特别感谢 Brian Kernighan、Don Knuth、Apostolos Lerios 和 Bob Martin，他们详细阅读了本书最初 Pascal 版本的手稿，并提出了很多宝贵意见。此外，我们已经收到 Michael Anderson、Margaret Johnson、Udi Manber、Joseph Naor、Prabhakar Ragde、Rocky Ross 和 Shuky Sagiv 对本书 Pascal 版本做出的课程测试报告，在此对他们表示由衷的感谢。

还有很多人找出了本书原稿以及 Pascal 版本各印次中的错误。为此，我们还要感谢以下人士：Susan Aho、Michael Anderson、Aaron Edsinger、Lonnie Eldridge、Todd Feldman、Steve Friedland、Christopher Fuselier、Mike Genstil、Paul Grubb III、Barry Hayes、John Hwang、Hakan Jakobsson、Arthur Keller、Dean Kelley、James Kuffner Jr.、Steve Lindell、Richard Long、Mark MacDonald、Simone Martini、Hugh McGuire、Alan Morgan、Monnia Oropeza、Rodrigo Philander、Andrew Quan、Stuart Reges、John Stone、Keith Swanson、Steve Swenson、Sanjai Tiwari、Eric Traut 和 Lynzi Ziegenhagen。

感谢 Geoff Clem、Jon Kettenring 和 Brian Kernighan 在筹备本书 C 语言版的过程中提出的建设性意见。

感谢 Peter Ullman 绘制了本书中的很多图片，感谢 Dan Clayton、Anthony Dayao、Mat Howard 和 Ron Underwood 在 TEX 字体方面提供的帮助，还要感谢 Hester Glynn 和 Anne Smith 在手稿准备阶段提供的帮助。

代码、勘误和注释的在线访问

大家可以匿名访问 FTP 主机 `ftp-cs.stanford.edu` 获取本书中主要程序的副本。请使用用户名 `anonymous`，并用该用户名加上主机名作为密码登录该 FTP。然后可以执行 `cd fcsc`，来找到书里出现的程序。我们还打算在该目录中保存勘误信息，以及可以提供的课程讲义。

Alfred V.Aho
于新泽西州查塔姆
Jeffrey D.Ullman
于加州斯坦福
1994 年 7 月

目　　录

第 1 章

计算机科学：将抽象机械化

计算机科学是个新领域，不过它几乎已经触及人类工作的每个方面。计算机、信息系统、文本编辑器、电子表格的普及，以及使得计算机更便于使用、人们生产效率的精彩应用程序的激增，都显示出计算机科学对社会的影响。该领域有个重要的部分，涉及如何让程序设计更容易以及让软件更可靠。不过从根本上讲，计算机科学是一门抽象的科学，它为人们思考问题以及找到适当的机械化技术解决问题而建立模型。

其他科学是顺其自然地研究宇宙。例如，物理学家的工作就是理解世界是如何运转的，而不是去创造一个用物理定律能更好地理解的世界。而计算机科学家则必须抽象现实世界中的问题，使其既可以为计算机用户所理解，又可以在计算机内加以表示和操作。

进行抽象的过程有时很简单。例如，我们能熟练地用"命题逻辑"这种抽象方式，为制造计算机所使用的电子电路的行为建模。通过逻辑表达式进行的电路建模是不准确的，它简化了或者说是抽象掉了很多细节，比如电子流经电路和门所花的时间。然而，命题逻辑模型已经足够帮助我们顺利设计计算机电路了。我们将在第12章中更多地探讨命题逻辑。

再举个例子，假设我们要为各种课程的期末考试排定时间。也就是说，我们必须为各门课程的考试指定时段，只有在没有学生同时选择某两门课程的前提下，才将这两门课程的考试安排在同一时段。如何为这一问题建模，起初可能不太好确定。一种方式是为每门课程画一个称为*节点*（node）的圆，如果有学生同时选择了两门课程，就画一条线来连接相应的两个节点，这条线称为*边*（edge）。图1-1表示了5门课程可能的关系图，这幅图就是*课程冲突图*。

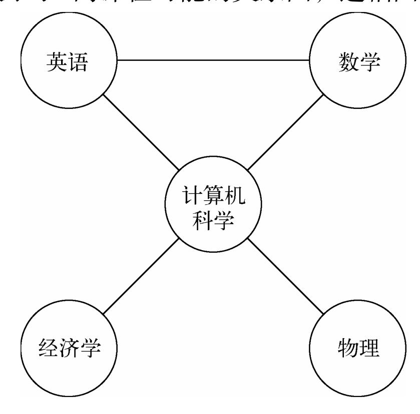


图1-1　5门课程的课程冲突图，两门课程之间的边表示至少有一个学生同时选择了这两门课程

有了课程冲突图，我们就可以通过在图中反复找出并删除“最大独立集”来解决考试安排问题。独立集是没有边相连接的节点的集合。如果不能再向某独立集添加图中的其他节点了，那么就说这个独立集是最大独立集。即，一个图中包含节点数目最多的独立集称为最大独立集。在说课程时，最大独立集就是指没有共同学生的课程的最大集合。在图1-1中，{经济学，英语，物理}就是一个最大独立集。最大独立集中的这些课程被指定到第一个时段。

我们从图中删除第一个最大独立集中的节点以及这些节点附带的边，接着在剩下的课程中找出最大独立集。下一个可选的最大独立集是单元素集{计算机科学}。这个最大独立集中的课程便被分配到第二个时段。

如此重复找出并删除最大独立集，直到课程冲突图中不再有任何节点。至此，所有课程都已经被分配到各时段中。本例中，在两次迭代之后，课程冲突图中就只剩下数学节点了，而它就组成了最后一个最大独立集，将被指定到第三个时段中。形成的考试排期如下：

时　段	课程考试
1	经济学，英语，物理
2	计算机科学
3	数学

这一算法不见得会将各门需要考试的课程分布在数目尽可能少的时段中，不过它很简单，而且生成的时间安排中所含的时段数目往往接近最小值。利用第9章介绍的技术，它也很容易被设计成计算机程序。

请注意，这种方式会将一些可能很重要的问题细节抽象掉。例如，它可能会让某个学生在5个连续的时段内参加5科考试。也许我们可以建立这样一个模型，对某个学生一次可能连续参加考试的科目数加以限制，不过这样一来，建立的模型和考试安排问题的解决方案都可能变得更加复杂。

抽象：不用担心

读者可能会对“抽象”这个词有所忌惮，因为我们都有这样一种直觉：抽象的东西都是难以理解的。例如，人们一般会认为抽象代数（研究群、环，诸如此类）要比高中时学的代数难。然而，我们所使用的抽象意味着简化，是将现实中复杂而详细的情景替换为解决问题所使用的可理解模型。也就是说，我们将那些对解决问题而言影响甚微或根本没有影响的细节“抽象掉”了，从而建立一个让我们能处理问题实质的模型。

通常情况下，找到好的抽象方式是相当困难的，因为计算机能执行的任务有限，执行速度也有限。在计算机科学的初期阶段，一些乐观主义者认为机器人很快就能像《星球大战》中的C3PO机器人那样神通广大。自那时起，我们已经了解到，要让计算机（或机器人）具有“智能”行为，就需要为计算机提供一个本质上跟人类所支配的世界一样详细的模型，不仅要包括事实（“萨莉的电话号码是555-1234”），还要包括原则和关系（“如果抛出某物体，它通常会向下坠落”）。

我们在“知识的表示”这一问题上已经取得了很大的进步，设计出了一些抽象方式，可用来构建进行某类推理的程序。有向图便是这种抽象的一个例子，它用节点表示实体（“猫”或“松

毛”），用从一个节点指向另一个节点的箭头（称为弧）代表关系（“松毛是只猫”，“猫是动物”，“松毛的牛奶碟子归松毛所有”），图1-2就展示了这样一幅图。

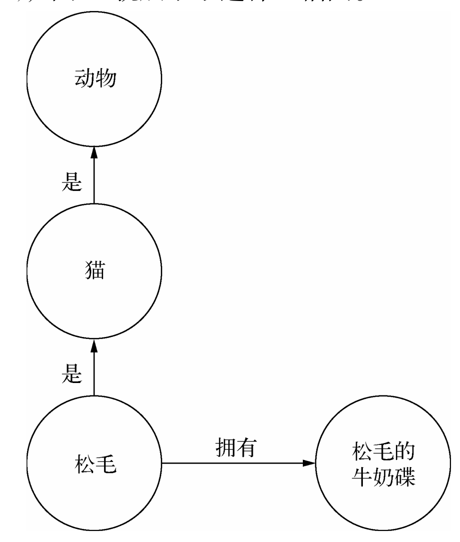


图1-2　这幅图用于表示与“松毛”相关的知识

另一种实用的抽象是形式逻辑，它让我们可以运用推理规则推导事实，比如“如果*X*是只猫，*Y*是*X*的母亲，那么*Y*是只猫”。不过，为现实世界或其关键部分建模（或者说是对其进行抽象）的过程，却仍是计算机科学所面临的一大挑战，是近期没法彻底解决的。

1.1　本书主要内容

本书目标读者应当具有一定的ANSI C语言程序设计实践经验，本书旨在为这些读者介绍计算机科学的基本概念和重点内容。书中强调了如下三种重要的问题解决工具。

(1) *数据模型*。数据特征的抽象，用来描述问题。我们已经提到了两种模型：逻辑和图，而在本书中还会看到很多其他的模型。

(2) *数据结构*。用来表示数据模型的编程语言结构。例如，C语言提供了内置的抽象，比如结构和指针，使我们能够构建数据结构，表示像图这类的复杂抽象。

(3) *算法*。操作用数据模型抽象、数据结构等形式表示的数据，从而获取解决方案的技术。

1.1.1　数据模型

我们在两种情况下会提到数据模型。像本章开头讨论的图这样的数据模型，是常用于协助形成问题解决方案的抽象。我们还会在本书中了解多种这样的数据模型，比如第5章介绍的树、第6章介绍的表、第7章介绍的集、第8章介绍的关系、第9章介绍的图、第10章介绍的有限自动机、第11章介绍的语法，以及第12章和第14章介绍的逻辑。

数据模型还与编程语言及计算机相关。比如，C语言的数据模型就包含诸如字符、多种长度的整数以及浮点数这类的抽象。C语言中的整数和浮点数只是数学意义上整数和实数的近似值，因为计算机所能提供的算术精度是有限的。C语言数据模型还包括结构、指针和函数这样的类型，我们将在1.4节中详细介绍。

1.1.2　数据结构

当手头问题的数据模型不能直接用编程语言内置的数据模型表示时，我们就必须使用该语言所支持的抽象来表示所需的数据模型。为此，我们研究了*数据结构*，将编程语言中没有显式包含的抽象，以该语言的数据模型表示出来。不同的编程语言可能有着大不相同的数据模型。例如，与C语言不同，Lisp语言直接支持树，而Prolog语言则内置了逻辑数据模型。

1.1.3　算法

*算法*是对可机械执行的一系列步骤精准而明确的规范。用来表示算法的可以是任何一种可被常人理解的语言，不过在计算机科学领域，算法多用编程语言正式地表现为计算机程序，或用编程语言混合英语语句的非正式风格来表示。大家在学习编程时很可能已经遇到过一些重要算法。例如，有不少为数组元素*排序*的算法，就是按照从小到大的顺序排列数组元素。有一些诸如二叉查找（binary searching）之类的查找算法很巧妙，可以通过反复将某给定元素在数组中可能出现的部分对半划分，迅速地找到这个元素。

这些算法以及其他一些解决常规问题的“招数”，是计算机科学家们在设计程序时会用到的工具。我们将在本书中学习诸多此类技巧，包括重要的排序和查找方法。此外，我们还要了解使一种算法优于其他算法的因素。很多时候，运行时间（running time），或者说算法处理输入所花的时间，是算法“质量”的重要一环，我们会在第3章中讨论。

算法的其他方面也很重要，特别是简易性。理想情况下，算法应该易于理解，并易于转变成可运转的程序。而且，懂得相应知识的人在阅读了实现该算法的代码后，应该能理解由该算法转变而来的程序。不过快速和简易往往是不能两全的，所以我们必须要明智地选择算法。

1.1.4　基本思路

在进一步阅读本书的过程中，我们将遇到一些重要的统一原则。在这里要提以下两点。

(1) *设计代数*。在底层模型得到充分了解的某些领域，我们可以提出一些表示法，以便表示和评价某些折衷的设计方案。通过这样的认识，我们可以提出一些设计理论以构建出设计良好的系统。命题逻辑，加上第12章中的*布尔代数*这种相关的表示法，就是设计代数的一个好例子。有了它，我们可以为数字计算机中的子系统设计高效的电路。其他设计代数的例子还包括第7章中的集代数、第8章中的关系代数，以及第10章中的正则表达式代数。

(2) *递归*。作为一种可用来定义概念和解决问题的实用技术，递归特别值得一提。我们会在第2章中详细讨论递归，本书后续内容中也会反复用到它。每当我们需要精确地定义对象，或需要解决问题时，都应该问一问：“递归解决方案应当是什么样子呢？”递归方案的简易和效率常使其成为最优方法。

1.2　本章主要内容

本章接下来的部分将为计算机科学的学习做好铺垫，要介绍以下主要概念。

- 数据模型（1.3节）。
- C语言的数据模型（1.4节）。
- 软件开发流程的主要步骤（1.5节）。

我们会介绍一些例子，讲讲抽象和模型出现在计算机系统的几种方式。其中特别提到了编程语言中的模型、特定系统程序（比如操作系统）中的模型，以及计算机所使用电路中的模型。由于软件是当今计算机系统的重要组成部分，因此我们需要理解软件开发流程、模型和算法扮演的角色，以及软件开发中计算机科学只能以有限方式解决的那些方面。

在1.6节中会介绍一些常规定义，它们在全书的C语言程序中都将用到。

1.3 数据模型

任何数学概念都可称为数据模型。而在计算机科学领域，数据模型通常包含以下两个方面。

(1) 对象可以采用的值。例如，很多数据模型包含具有整数值的对象。数据模型的这个方面是静态的，它告诉我们对象能接受哪些值。编程语言数据模型的这一静态部分通常被称为*类型系统*。

(2) 数据的运算。例如，我们常常会对整数执行加法这样的运算。模型的这一方面是*动态*的，它告诉我们改变值和创建新值的方式。

1.3.1 编程语言数据模型

每种编程语言都有自己的数据模型，这些数据模型互不相同，而且通常有相当大的差异。多数编程语言处理数据所遵循的基本原则是，每个程序都可以访问我们用于表示存储区域的“框”。每个框都具有一个类型，比如`int`或`char`。框中可以存储类型对应的值，通常将可以存储到这些框中的值称为*数据对象*。

我们还要为这些框命名。一般来说，框的名称可以是任何指示该框的表述性词语。我们通常会将框的名称视作该程序的变量，不过情况并非完全如此。例如，如果x是递归函数F的局部变量，那么就可能会有很多名为x的框，每个x都与对F的不同调用相关联。这样的话，这种框的真实名称就是x与对F的某次调用的组合。

C语言中的多数数据类型都是我们熟悉的：整数、浮点数、字符、数组、结构和指针。这些都是静态的概念。

可以对数据进行的操作包括整数和浮点数的常规算术运算、数组或结构元素的存取操作，以及指针的*解引用*（也就是找到指针所指向的元素）。这些运算都只是C语言数据模型动态部分的一部分。

在程序设计课程中，我们可能会看到C语言中不包括的重要数据模型，比如表、树和图。用数学语言来讲，表就是可以写成$(a_1, a_2, \cdots, a_n)$这种形式的n个元素组成的序列，其中a_1是第一个元素，a_2是第二个，以此类推。表的运算包含插入新元素、删除元素，以及*拼接*表（也就是将一个表追加到另一表的末端）。

✦ 示例 1.1

在C语言中，整数表可以用*链表*这种数据结构表示，表的元素被存储在链表的节点中。链表及其节点可用如下类型声明定义。

```
typedef struct CELL *LIST;
struct CELL {
    int element;
    struct LIST next;
};
```

该声明定义了有着两个字段的自引用结构CELL。第一个字段是element，存放着表中元素的值，而且其类型为int。

每个CELL的第二个字段是next，存放着指向节点的指针。请注意，LIST类型其实是指向CELL的指针。因此，CELL类型的结构可以通过它们的next字段链接起来，构成我们通常所说的链表，如图1-3所示。next字段既可以被视为指向下一个节点的指针，也可以代表从某节点起的整段链表。同理，整个链表也可以用指向链表第一个单元的LIST类型的指针表示。

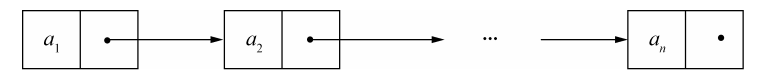


图1-3　表示表$(a_1, a_2, \cdots, a_n)$的链表

单元是用长方形表示的，其左边部分表示元素，右边部分存放指针（表示为指向下一个单元的箭头）。存放指针的方框中的点表示该指针为NULL[①]。第6章将更详细地介绍表。

数据模型与数据结构

尽管名称类似，但“表”和“链表”却是非常不同的概念。表是种数学抽象，或者说是数据模型。而链表则是种数据结构，是通常用于C语言及相似语言中的数据结构，用来表示程序中的抽象表。而有些编程语言则不需要用数据结构来表示抽象表。例如，表$(a_1, a_2, \cdots, a_n)$在Lisp语言中可以直接表示为$[a_1, a_2, \cdots, a_n]$，而在Prolog语言中也可以表示为类似形式。

1.3.2　系统软件的数据模型

数据模型不仅存在于编程语言中，而且存在于操作系统和应用程序中。大家可能熟悉UNIX或MS-DOS这样的操作系统，也可能熟悉Microsoft Windows。[②]操作系统的功能是管理和调度计算机的资源。像UNIX这样的操作系统，其数据模型具有文件、目录和进程这样的概念。

(1) 数据本身存储在文件中，在UNIX系统中，文件都是字符串和字符。

(2) 文件被组织成目录，目录就是文件和（或）其他目录的集合。目录和文件形成了树形结构，而文件处在树叶的位置[③]。图1-4中的树可以表示UNIX操作系统的目录结构。目录是用圆圈表示的。根目录/包含名为mnt、usr、bin等的目录。目录/usr含有目录ann和bob，而目录ann下含有3个文件：a1、a2和a3。

(3) 进程是指程序的独立执行。进程接受流作为输入，并产生流作为输出。在UNIX系统中，进程可以通过管道连接，让一个进程的输出作为下一个进程的输入。这种进程组合可看作有着自己输入输出的独立进程。

---

① NULL是标准头文件stdio.h中定义的符号常量，用来表示未指向任何内容的指针的值。本书中的NULL指针都作此义解释。

② 如果对操作系统不熟悉，那么可以跳过下面几个段落。不过大多数读者都应该接触过操作系统，可能只是称呼不同。例如，Macintosh“系统”就是一种操作系统，只是使用了不同的术语。例如，在苹果用语中，目录就被称为“文件夹”。

③ 不过，目录中的“链接”可能会让某个文件或目录看起来像是几个不同目录的一部分。

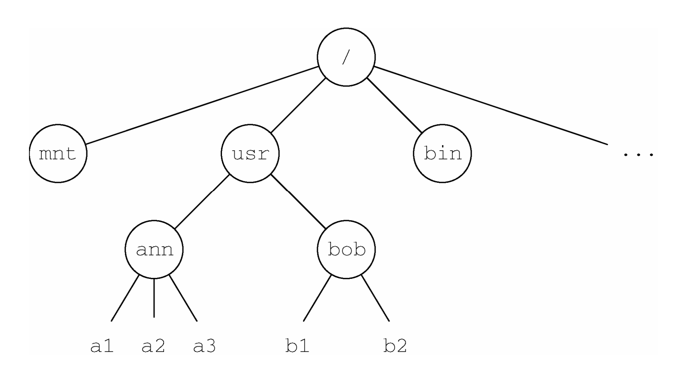


图1-4　具有代表性的UNIX目录/文件结构

✦　示例 1.2

想想如下UNIX命令行。

```
bc | word | speak
```

符号 | 表示管道，该操作使符号左边进程的输出成为符号右边进程的输入。程序bc是桌面计算器，接受算术表达式（例如2+3）作为输入，并生成答案5作为输出。程序word用来将数字转换成单词，而speak则将单词转换成音素序列，接着通过扬声器将语音合成器合成的声音播放出来。这三个程序通过管道连接起来，使这条UNIX命令行成为了一个进程，并表现为一个“会说话的”桌面计算器。它接受算术表达式作为输入，并产生说出来的答案作为输出。本示例还可以说明，将复杂的任务处理成多个简单功能的组合，实现起来可能会更加简单。

操作系统还有其他许多方面，比如它如何控制数据安全以及与用户的互动。不过，即便是通过这些简单的观察，也应该很容易看出，操作系统的数据模型和编程语言的数据模型是相当不同的。

文本编辑器中有另一种数据模型。文本编辑器的每种数据模型都结合了文本字符串的表示和对文本的编辑操作。这种数据模型通常会包含行的概念，行和多数文件一样，就是字符串。不过，与文件不同的是，行可能有着与其相关联的行号。行还可能被组织成更大的单元（比如段落），而且对行进行的操作通常适用于行内的任何位置，而不会像多数常见的文件操作那样，只是对前部进行操作。一般的文本编辑器会支持“当前”行（光标所在的那一行）的概念，还可能支持行内当前位置的概念。文本编辑器执行的操作包括对行的多种修改，比如在行内删除或插入字符、删除行，以及创建新行。在一般的文本编辑器中还可以在已编辑文件的行中搜索特定的字符串。

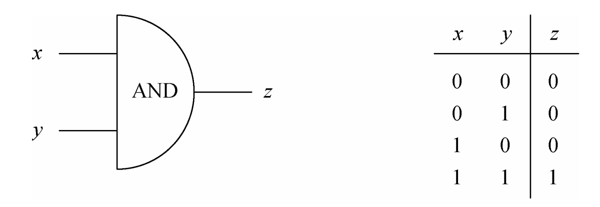
其实，如果看看其他熟悉的软件，比如电子表格或视频游戏，就会发现，每个调用程序都必须遵守被调用程序的数据模型。我们见到的各种数据模型通常彼此间截然不同，无论是用来表示数据的原语，还是向用户提供的数据操作方式，全都不同。而且各数据模型都是通过数据结构和使用它们的程序，用某种编程语言实现的。

1.3.3　电路的数据模型

在本书中我们还会看到计算机电路使用的数据模型。这种模型就是*命题逻辑*，在计算机设计中是最实用的。计算机是由称为门的基本元件组成的。每个门都有着一个或多个输入以及一

个输出，输入或输出的值只能是0或1。门具有一个简单的功能，比如AND运算（与运算），就是如果所有输入为1，那么输出就是1，而如果至少有一个输入为0，那么输出就是0。从某个抽象层次来讲，计算机设计就是选择如何连接门来执行计算机基本运算的过程。当然也存在其他很多与计算机设计相关的抽象层次。

图1-5展示了常见的与门符号以及对应的*真值表*，该表指明了每对输入值搭配经过该门产生的输出值[①]。我们将在第12章中介绍真值表，并在第13章中介绍门及门的互相连接。



x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

图1-5　与门及其真值表

✦ 示例 1.3

执行C语言赋值语句a=b+c，计算机会使用*加法*电路执行加法运算。在计算机中，所有数字都是以二进制的形式，使用0和1这两个数字（叫作*二进制数字*，或简称*位*）表示的。二进制加法计算也遵守十进制加法的法则，从右端的数字开始相加，如果产生进位，就将进位加到右起第二位上，如果这一位上相加的结果还产生进位，就继续加到右起第三位上，以此类推。

我们可以用几个门来组建一位加法器（one-bit adder）电路，如图1-6所示。两个输入位x和y，一个*进位输入位*c，经过相加，形成一个和值位z，以及*进位输出位*d。更精确地讲，如图1-7所示，如果c、x和y中有不少于两个的值为1，那么d的值就是1，而如果c、x和y中有奇数个（1个或3个）的值为1，那么z的值就是1。进位输出位后面跟上和值位（即dz）就形成了一个两位的二进制数，这就是x、y和c为1时的总值。在这种情况下，这个一位加法器就完成了输入的相加运算。

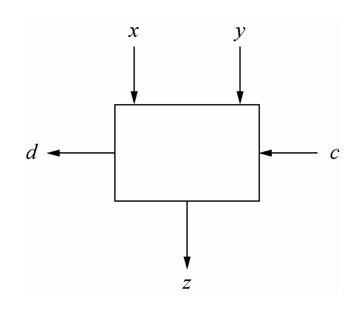


图1-6　一位加法器：dz是$x + y + c$的和

① 请注意，若我们将1视为“真”，将0视为“假”，则与门执行的是和C语言中&&运算符相同的逻辑运算。

x	y	c	d	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

图1-7 一位加法器的真值表

行波进位加法算法

在进行十进制数的加法时，我们都使用过行波进位算法。拿456+829举例，相加的步骤应该如下所示。

```
  1           0
  4 5 6       4 5 6         4 5 6
  8 2 9       8 2 9         8 2 9
 --------    --------     ---------
      5         8 5       1 2 8 5
```

也就是说，第一步，我们会将最右边的位相加，$6+9=15$。记下5，并将进位1放到第二列。第二步，我们将进位输入1与右起第二位的两个数字相加，得到$1+5+2=8$。记下8，进位是0。第三步，将进位输入0，与右起第三位上的数字相加，得到$0+4+8=12$。记下2，由于我们已经计算到了最左边的位，因此就不将1进位，而是将其作为结果中最左边的一位。

二进制行波进位加法也有着相同的原理。只不过，在每一位上，进位和要相加的“数字”要么是0，要么是1。因此一位加法器完整地描述了单个数位上的加法表。也就是说，如果三个位都是0，那么和就是0，就记下0以及进位0。如果三个位中有一个是1，那么和就是1，就记下1及进位0。如果三个位中有两个是1，那么和就是2，也就是二进制数10，就记下0以及进位1。如果三个位全是1，那么和就是3，也就是二进制数11，就记下1及进位1。例如，用行波进位加法将二进制数101和111相加的步骤如下所示。

```
  1           1
  1 0 1       1 0 1         1 0 1
  1 1 1       1 1 1         1 1 1
 --------    --------     ---------
      0         0 0       1 1 0 0
```

很多计算机用32位数字来表示整数。所以加法器电路可由32个一位加法器组合而成，如图1-8所示。该电路通常称为行波进位加法器，因为进位是从右向左一次一位行进的。要注意，最右侧（最低位）一位加法器的进位总是0。位序列$x_{31}x_{30}\cdots x_0$表示一个加数，而$y_{31}y_{30}\cdots y_0$则表示另一个加数。和就是$dz_{31}z_{30}\cdots z_0$，也就是说，和的第一位是最左侧一位加法器的进位输出，而接下来的位就是从左往右各加法器的和值位。

如图1-8所示的电路是由位数据模型以及门的原始运算形成的算法。不过这不是一种特别好的算法，因为要是不计算完最右侧那一位，就不能计算z_1或右起第二位的进位输出。不计算完

右起第二位，就不能计算z_2或右起第三位的进位输出，诸如此类。因此，该电路花费的时间就是加数的位长（在本例中是32）乘上每个一位加法器执行运算所需的时间。

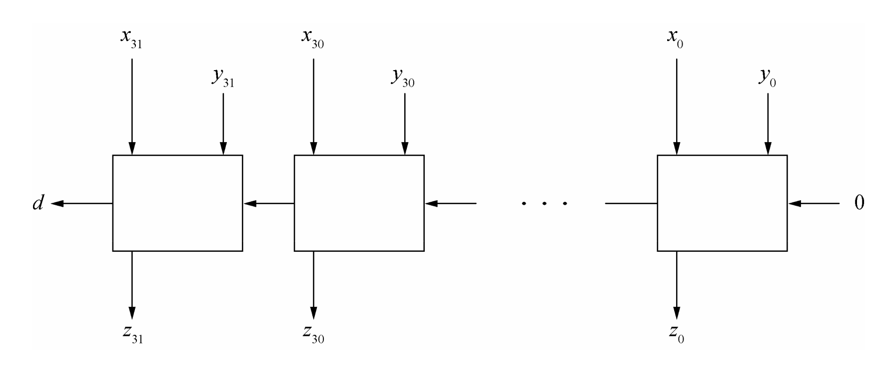


图1-8　行波进位加法器：$dz_{31}z_{30}\cdots z_0=x_{31}x_{30}\cdots x_0+y_{31}y_{30}\cdots y_0$

有人可能会认为，进位在每个一位加法器间"行进"的需求，是加法定义中固有的。要是这些读者知道计算机还有快得多的加法计算方式，肯定会大吃一惊的。我们将在第13章讨论电路的设计时，介绍一种改进过的加法算法。

1.3.4　习题

(1) 解释数据模型静态方面和动态方面的差异。

(2) 描述自己最喜欢的视频游戏的数据模型。区分其模型的静态方面和动态方面。提示：静态部分不仅是指计分牌上不会移动的部分。例如，在《吃豆人》游戏里，静态部分不仅包括地图，还包括"强化药丸"和"怪物"等。

(3) 描述自己最喜欢的文本编辑器的数据模型。

(4) 描述电子表格程序的数据模型。

1.4　C 语言数据模型

在本节中，我们将重点介绍C语言所使用数据模型的重要部分。以图1-9所示的C语言程序为例，该程序使用变量num来计算其输入中所含的字符数。

```
#include <stdio.h>
main()
{
    int num;
    num = 0;
    while (getchar() != EOF)
        ++num; /* add 1 to num */
    printf("%d\n", num);
}
```

图1-9　计算输出所含字符数的C语言程序

程序的第一行告诉C语言预处理器，将标准输入/输出文件`stdio.h`包含为源的一部分。该文件含有`getchar`及`printf`函数的定义，以及表示文件结束的符号常量`EOF`。

C语言程序本身也包含一系列的定义，既可以是函数的定义，也可以是数据的定义，其中必须要有`main`函数的定义。图1-9所示程序的函数体中，第一条语句声明了`int`类型的变量`num`（在C语言程序中，所有变量在使用前都必须先声明），下一条语句将`num`初始化为`0`，接下来的`while`语句则使用库函数`getchar`一次读入一个输入字符，并在每次字符读入后递增`num`变量，直到没有输入字符可供读入为止。输入中的特殊值`EOF`会提示文件已达末尾。`printf`语句则会将`num`的值以十进制整数之后加上换行符的形式打印出来。

1.4.1　C语言类型系统

首先介绍C语言数据模型的静态部分——类型系统，它描述了数据可能拥有的值。随后要讨论C语言数据模型的动态部分，也就是可以对数据进行的操作。

在C语言中，有着类型构成的无限集合，其中的任意元素都可以成为与某个特定变量相关联的类型。这些类型以及构成类型的规则就形成了C语言的*类型系统*。类型系统包含整数这样的基本类型以及一些类型构成规则（type-formation rule），利用这些规则，我们可以用已知的类型逐步构建更为复杂的类型。C语言的基本类型包括：

(1) 字符（`char`、`signed char`、`unsigned char`）；

(2) 整数（`int`、`short int`、`long int`、`unsigned`）；

(3) 浮点数（`float`、`double`、`long double`）；

(4) 枚举（`enum`）。

整数和浮点数称为*算术类型*。

类型构成规则假设我们已经有了一些类型，可以是基本类型或使用这些规则构建好的其他类型。以下是C语言中的一些类型构成规则。

(1)*数组类型*。可以用以下声明构建一个元素类型为T的数组：

```
T A[n]
```

该语句声明了包含n个元素的数组`A`，其中每个元素都是T类型的。在C语言中，数组下标是从0开始的，所以数组的第一个元素是`A[0]`，而最后一个元素是`A[n-1]`。数组可由字符、算术类型、指针、结构体、共用体或其他数组构成。

(2) *结构体类型*。在C语言中，结构体是由称为成员或字段的变量构成的分组。在结构体中，不同的成员可以具有不同的类型，但每个成员都必须具有某一个类型的元素。如果T_1、T_2、…、T_n是类型，而M_1、M_2、…、M_n是成员名称，那么如下声明

```
struct S {
    T1 M1;
    T2 M2;
    …
    Tn Mn;
}
```

就定义了*标记*（即其类型的名称）为S而且具有n个成员的结构体。对$i = 1$、2、…、n来说，第i个成员名称为M_i，且其值为T_i类型。示例1.1就展示了一个结构体。该结构体的标记是`CELL`，并含有两个成员。第一个成员的名称是`element`，类型为整数。第二个成员名称为`next`，它的类型是指向某个同类型结构体的指针。

结构体标记S是可选的，不过它可以在随后的声明中为表示类型提供方便的简写。例如，声明

```
struct S myRecord;
```

定义了变量myRecord是一个类型为*S*的结构体。

(3) 共用体类型。共用体类型允许一个变量在程序执行的不同时期具有不同的类型。声明

```
union{
    T1 M1;
    T2 M2;
    …
    Tn Mn;
} x;
```

定义了变量x，可以存放类型为T_1、T_2、…、T_n中任意一种的值。成员名称M_1、M_2、…、M_n用来指示x的值现在应该是哪种类型。也就是说，x.M_i就表明x的值是类型为T_i的值。

(4) 指针类型。C语言的独特之处在于对指针的依赖。指针类型的变量包含某个存储区域的地址。可以通过指针，间接地访问另一个变量。声明

```
T *p;
```

定义了变量p是指向某个*T*类型变量的指针。用p来表示指向*T*的类型指针的框，框p的值就是个指针。我们往往将p的值表示成一个箭头，而不是将其表示成*T*类型的对象本身，如图1-10所示。真正出现在p框中的是*T*类型对象在计算机中存储的地址（或位置）。

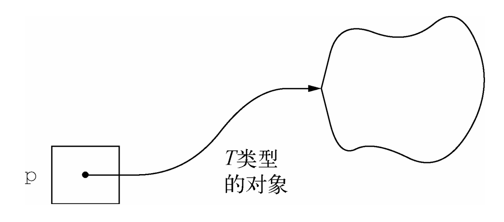


图1-10　变量p是指向*T*的类型指针

考虑如下声明

```
int x, *p;
```

在C语言中，一元运算符&是用来获取对象地址的，所以声明

```
p = &x;
```

将x的地址赋值给p，也就是说，这让p指向x。

用在p前面的一元运算符*会获取p指向的框的值，所以声明

```
y = *p;
```

会将框p指向的内容赋值给y。如果y是int类型的变量，那么

```
p = &x;
y = *p;
```

就等价于赋值语句

```
y = x;
```

✦ 示例 1.4

C语言的typedef结构可用来创建类型名称的同义字。

看一看图1-11中的4个typedef声明。依照对C语言中数据的传统看法，类型type1是有10个槽（slot）的数组，每个槽中都存放着一个整数，如图1-12a所示。同样，类型type2的对象是指向这类数组的指针，如图1-12b所示。而类型type3的结构体则被表现为图1-12c中所示的形式，每个字段都有一个槽与其对应。请注意，字段名称（例如field1）实际上并未与字段的值一起出现。最后，数组类型type4的对象将会有5个槽，每个槽都存放着类型type3的对象，即如图1-12d所示的结构体。

```
typedef int Distance;
typedef int type1[10];

typedef type1 *type2;

typedef struct {
    int field1;
    type2 field2;
} type3;

typedef type3 type4[5];
```

图1-11　一些C语言typedef声明

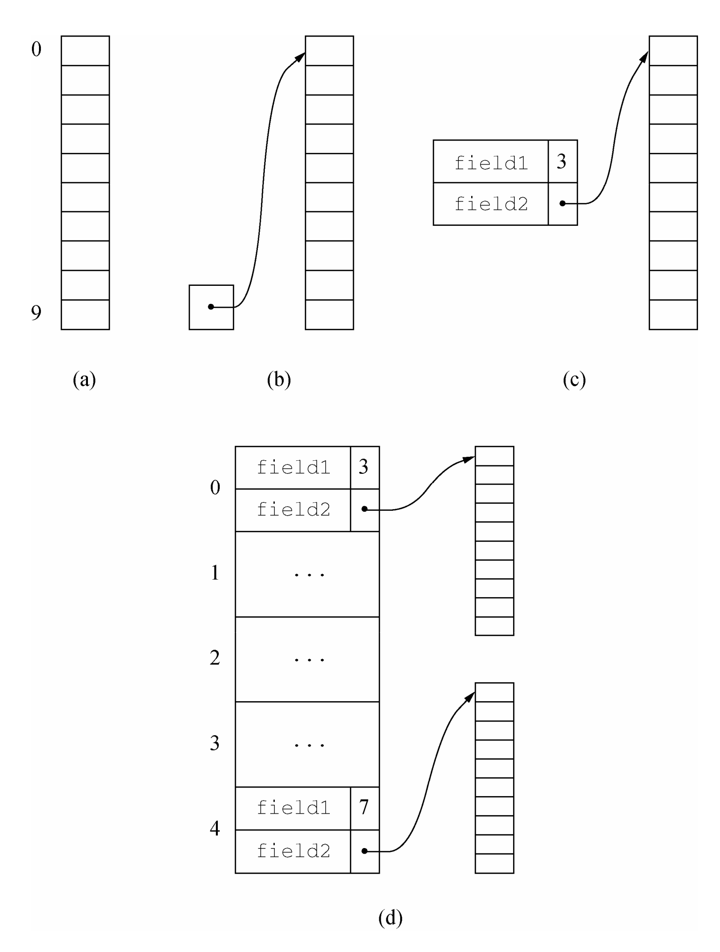


图1-12　图1-11中类型声明的形象化表示

类型、名字、变量和标识符

与数据对象相关的一些术语有不同的含义却又容易混淆。首先，类型描述了数据对象的“形状”。在C语言中，可以使用typedef结构为已有的类型定义一个新名字T。

```
typedef <类型描述符> T
```

这里的类型描述符是个表达式，告诉我们T类型的对象是什么样子。

类型为T的typedef声明实际上并没有创建T类型的对象。要创建T类型的对象，需要使用如下形式的声明

```
T x;
```

这里的x是个标识符，或者说是“变量名”。x有可能是静态的（不是任何函数的局部变量），在这种情况下，表示x的框在程序开始时就创建了。如果x不是静态的，那么它应该是某个函数F的局部变量。在调用F时，就会创建一个名为“与本次对F的调用相关联的x”的框。更准确地说，该框的名称还是x，不过只在执行本次对F的调用时，才使用标识符x来表示该框。

正如文中提到的，因为F可能是递归函数，所以可能存在许多名称涉及标识符x的框。甚至可能会有其他函数使用标识符x命名自己的某个变量。此外，名字比标识符更具一般性，因为有很多种表达式可以用来为框命名。例如，我们提到过*p可以是指针p指向的某个对象的名字，而该对象的其他名字也可以是复杂的表达式，比如(*p).f[2]或p->f[2]。这两个复杂表达式是等价的，都表示指针p指向的结构体中f字段数组的第二个元素。

1.4.2　函数

函数也具有与之关联的类型，即使我们没有像处理程序变量那样，将框或“值”与函数相关联。对任意的一列类型T_1、T_2、…、T_n，我们可以定义一个函数，具有n个类型依次为这些类型的参数。这一列类型后面带上函数返回的值（返回值）的类型，就是这个函数的“类型”。如果函数没有返回值，那么该函数就是void类型的。

一般情况下，可以应用类型构成规则任意地构建类型，不过也存在一些限制。比如，不能构建“函数数组”，不过构建由指向函数的指针构成的数组是可以的。在C语言中构建类型的完整规则可以在ANSI标准中找到。

1.4.3　C语言数据模型中的操作

C语言数据模型中的数据操作可分为以下三类。

(1) 创建或销毁数据对象的操作。

(2) 访问或修改数据对象某些部分的操作。

(3) 将若干数据对象的值组合起来，为某个数据对象生成新值的操作。

1.4.4　数据对象的创建和销毁

对于数据的创建，C语言提供了几种简陋的机制。在函数被调用时，会创建对应每个局部参数的框，这些框都用来存放参数的值。

另一种数据创建机制是使用程序库例程malloc(n)，该例程可以返回一个指针，指向n个

未使用的连续字符位置，这些存储空间可被malloc的调用者用来存储数据。然后就可以在这一存储区域中创建数据对象。

C语言有着类似的方法来销毁数据对象。当函数返回时，该函数调用的局部参数将不复存在。例程free会释放malloc创建的存储空间。特别要说的是，调用free(p)的效果是释放p指向的存储区域。若使用free去销毁不是通过调用malloc创建的对象，会造成灾难性后果。

1.4.5 数据的访问和修改

C语言具有访问对象某些部分的机制。可以使用a[i]访问数组a的第*i*个元素，用x.m访问结构x的成员m，还可以用*p访问指针p指向的对象。

在C语言中，修改（或者说是写）值主要是由赋值运算符完成的，这让我们可以改变对象的值。

✦ 示例 1.5

如果变量a的类型是示例1.4中所定义的type4，那么

```
(*a[0].field2)[3] = 99;
```

就把值99赋给了数组a第一个元素所代表的结构体中field2指向的数组的第4个元素。

1.4.6 数据的组合

C语言有着丰富的运算符，可用来对值进行操作和组合。主要运算符包括如下这些。

(1) *算术运算符*。C语言提供了以下几种算术运算符。

(a) 用于整数和浮点数的常规二元算术运算符+、−、*、/。整数除法会取整（4/3得1）。

(b) 一元的+和−运算符。

(c) 取模运算符i%j的结果是*i*除以*j*的余数。

(d) 递增和递减运算符++和--，适用于单个整数变量反复从自身增加或减去1。这些运算符可以出现在它们的操作数之前，也可以出现在它们的操作数之后，取决于我们是想在改变变量的值之前还是之后计算该表达式的值。

(2) *逻辑运算符*。C语言中没有布尔类型，它使用“0”来表示逻辑值“假”，使用“非0”表示逻辑值“真”。[①]C语言使用以下几种逻辑运算符。

(a) &&表示AND运算。例如，表达式x&&y在两个操作数都非0的情况下会返回1，否则返回0。不过，如果x的值为0，就不考虑y的值了。

(b) ||表示OR运算。表达式x||y在x或y非0的情况下会返回1，否则返回0。不过，如果x的值非0，就不考虑y的值了。

(c) 一元的否定运算符!x在*x*非0时返回0，在*x*=0时返回1。

(d) 条件运算符是三元（三参数）运算符，用一个问号和一个冒号表示。表达式x?y:z在x为真（即*x*为非0）的情况下会返回y的值，在x为假（即*x*=0）的情况下会返回z的值。

(3) *比较运算符*。对整数或浮点数使用6种关系比较运算符之一（==、! =、<、>、<=、和>=），如果关系不成立，结果就为0，否则结果为1。

---

① 我们将反复使用TRUE和FALSE作为已定义的常量1和0，来表示布尔值，详见1.6节。

(4) 位运算运算符。C语言提供了一些实用的位逻辑运算符，将整数当作与它们的二进制形式相同的位字符串。这些运算符包括，用于按位与的&，用于按位或的|，用于按位异或的^，用于左移位的<<，用于右移位的>>，以及用于左移位的波浪字符（~）。
(5) 赋值运算符。C语言使用=作为赋值运算符。除此之外，还允许将x=x+y；这样的表达式写为x += y;这样的简短形式。类似的格式也可以用于其他二元算术运算符。
(6) 强制转换运算符。强制转换是指将某个类型的值转换成另一个类型的等价值的过程。例如，如果x是浮点数，而i是整数，那么x = i会导致i的整数值被转换成值相等的浮点数。在这里，强制转换运算符并未显式出现，不过C语言编译器会推断从整数到浮点数的转换是必要的，并自动执行所需的转换步骤。

1.4.7　习题

(1) 解释C语言程序的标识符与名字（用于“框”或数据对象）之间的区别。
(2) 举例说出有多个名字的C语言数据对象。
(3) 如果熟悉C语言之外的编程语言，描述一下它的类型系统和操作。

1.5　算法和程序设计

对数据模型、它们的属性及其适当用途的研究是计算机科学的一大核心，而与其同等重要的一大核心便是对算法以及与其相关的数据结构的研究。我们需要了解执行常见任务的最佳方法，而且需要学习设计优秀算法的主要技术。此外，我们还需要了解如何将数据结构和算法的使用融入创建实用程序的过程中。数据模型、算法、数据结构，以及它们在程序中的实现，这些主题相互依存，而且每个主题都会在本书中出现多次。在本节中，我们将粗略地提到一些与程序的设计和实现有关的知识。

1.5.1　软件的创建

在程序设计课上，当我们拿到编程问题时，可能需要设计解决问题的算法、用某种语言实现该算法、编译程序并用一些示例数据运行它，然后提交该程序给老师打分。

而在商业背景中，编程环境则完全不同。算法通常只不过是完整程序的一小部分，至少对那些简单平常到信手可拾的算法来说是这样。而程序通常是涉及硬件和软件的更大系统的组件。程序及其所嵌入的完整系统，都是由程序员和工程师团队开发的，这样的团队可能有数百人的规模。

软件系统的开发过程通常要跨越多个阶段。虽然这些阶段表面上可能和解决课堂编程任务所涉及的步骤有相似之处，但是构建软件系统来解决特定问题的功夫多数并没有花在编程上。下面要讲的是一种理想化的场景。

问题的定义和需求说明。在创建软件系统的过程中，最难也是最重要的部分是定义真正的问题所在并指明解决问题所需的条件。通常，问题的定义始于对用户需求的分析，不过这些需求通常是不准确的，而且很难写下来。系统架构师可能要咨询系统未来的用户，并对需求说明进行迭代，直到详解者（specifier，拟定需求说明的人）和用户都对定义和解决手头问题的需求说明感到满意为止。在需求说明阶段，为最终系统建立简单的原型或模型是有好处的，因为这样可以深入了解系统的行为和可能的用途。数据建模也是问题定义阶段的一个重要工具。

设计。一旦完成需求说明，系统的上层设计就已成形，而且主要组成部分也确定了。开发人员会拟定一份概述上层设计已完成的文档，文档中还可能包含系统的性能要求。该阶段还可能引入有关某些主要组件的更详细的需求说明。高性价比的设计往往需要重用或修改以前构造的组件，诸如面向对象技术这样的多种软件方法论推动了组件的重用。

实现。一旦敲定设计，就可以开始实现组件了。本书中讨论的很多算法都能在实现新组件的过程中派上用场。一旦完成组件的实现工作，就要对其进行一系列的测试，以确保它能像需求说明所说的那样工作。

集成和系统测试。当组件得到实现而且已经单独测试过，就应该将整个系统组合起来并进行测试。

安装和现场测试。一旦开发人员觉得系统已经能以令客户满意的状态运转，就可以将系统安装到客户的办公地点，并进行最终的现场测试。

维护。至此，我们可能会认为已经完成了大部分的工作。然而，还需要有维护工作。在很多情况下，维护可能要占据超过一半的系统开发成本。维护可能涉及修改组件来消除不可预见的副作用、修正或提高系统性能，或增加新功能等目的。因为维护是软件系统设计中很重要的部分，所以编写的程序务要正确、耐用、高效、可修改，并且能从一台计算机移植到另一台计算机。

尽早地发现错误很重要，最好是在问题定义阶段就能发现错误。越到后面的阶段，修复设计错误或编程错误的成本越高，对需求和设计的独立审查有利于减少后续的错误。

1.5.2　编程风格

编写他人能够轻松阅读和修改的程序，便能够显著减轻维护负担。好的编程风格都是练习的结果，建议大家一开始就试着编写方便他人理解的程序。没有什么神奇公式能确保程序的可读性，不过还是有一些实用经验可介绍给大家。

(1) 将程序分成相关的模块。

(2) 为程序排版，使其结构清晰。

(3) 编写易于理解的注释来解释程序。清晰准确地描述底层数据模型、用来表示数据模型的数据结构和每个例程所执行的操作。在描述例程时，要陈述对其输入作出的假设，并讲清输出和输入有什么关系。

(4) 对例程和变量使用有意义的名称。

(5) 尽可能避免使用明确的常数。例如，不要用数字7表示小矮人的个数，而是要使用诸如`NumberOfDwarfs`这样定义的常量，这样一来，如果决定再加上一个小矮人，就可以很方便地将该常量的值改为8。

(6) 避免使用“全局变量”，即不要为整个程序定义变量，除非程序中的大多数例程都要使用该变量所表示的数据。

另一个编程好习惯就是拥有成套测试输入，可以在编程时对每行代码进行测试。每当为程序增加了新功能，就可以运行这套测试，以确保新程序在处理这些起作用的输入时能和老程序行动一致。

1.6　本书中用到的一些 C 语言约定

在说明与C语言程序相关的概念时，有一些实用的定义和约定。其中一些是在标准头文件

stdio.h中也能找到的常规约定，而另一些则是为本书特别定义的，必须在使用它们的C语言程序中包含这些约定。

(1) 标识符NULL是指针的值，可能在任何出现指针的地方出现，但它是个不能指向任何内容的值。因此，出现在示例1.1链表节点的next字段中的NULL，可以用来表示链表的结尾。我们还将看到NULL在其他的数据结构中也有着诸多类似的用途。NULL在stdio.h头文件中得到了恰当的定义。

(2) 标识符TRUE和FALSE按如下方式定义

```
#define TRUE 1
#define FALSE 0
```

因此，在任何需要逻辑值“真”的情况中都可以使用TRUE，而在逻辑值为“假”的情况中都可以使用FALSE。

(3) 类型BOOLEAN被定义为

```
typedef int BOOLEAN;
```

在强调要表示的是表达式的逻辑值而非数值时，就会使用BOOLEAN。

(4) 标识符EOF是getchar()这样的文件读操作函数在无法继续从文件读出字节时返回的值。stdio.h文件为EOF定义了一个合适的值。

(5) 我们还要定义一个宏，用来生成示例1.1中所用节点的声明。图1-13就展示了一种可取的定义。它声明单元具有两个字段：element字段的类型是由参数Type给定的，而next字段则指向具有本结构的单元。该宏提供了两项外部定义：CellName是该类型结构体的名字，而ListName则是指向这些单元的指针的类型名称。

```
#define DefCell(EltType, CellType, ListType)            \
typedef struct CellType *ListType;                      \
struct CellType {                                       \
    EltType element;                                    \
    ListType next;                                      \
}
```

图1-13　用来定义表中单元的宏

✦ 示例 1.6

通过使用宏

```
DefCell(int, CELL, LIST);
```

可以定义示例1.1中那种类型的单元。

该宏随后会扩展为

```
typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
}
```

这样一来，我们就可以使用CELL作为整数单元的类型，并使用LIST作为指向这些单元的指针的类型。例如

```
CELL c;
LIST L;
```

定义了单元c，以及指向单元的指针L。请注意，通常会用指向表第一个单元的指针来表示一列单元，如果表为空，则用NULL来表示。

1.7 小结

至此，大家应该已经从本章中了解到以下概念。

- 数据模型、数据结构和算法是怎样用来解决问题的。
- 数据模型“表”和数据结构“链表”之间的差别。
- 无论是编程语言、操作系统，还是应用程序，每种软件系统中都存在着某种类型的数据模型。
- C语言所支持数据模型的关键要素。
- 大型软件系统开发过程的主要步骤。

1.8 参考文献

Kernighan and Ritchie [1988]是C语言的经典参考书。Roberts [1994]对使用C语言的程序设计进行了很好的介绍。

Stroustrup [1991]创造了C语言的面向对象扩展版——C++，C++已被广泛用于系统的实现。Sethi [1989]对几种主要编程语言的设计模型进行了介绍。

Brooks [1974]有力地描述了大型软件系统开发中存在的技术难题和管理难题。Kernighan and Plauger [1978] 针对改善编程风格提出了一些忠告。

American National Standards Institute(ANSI) [1990]. *Programming Language C*, American National Standards Institute, New York.

Brooks, F. P. [1974]. *The Mythical Man Month*, Addison-Wesley, Reading, Mass.

Kernighan, B. W.,and P. J. Plauger [1978]. *The Elements of Programming Style, second edition*, McGraw-Hill, New York.

Kernighan, B.W.,and D. M. Ritchie [1988]. *The C Programming Language,* second edition, Prentice-Hall, Englewood Cliffs, New Jersey.

Roberts, E .S. [1994].The Art and Science of C: A Library Based Introduction to Computer Science,Addison-Wesley, Reading, Mass.

Sethi, R. [1989]. *Programming Languages: Concepts and Constructs*, Addison-Wesley, Reading, Mass.

Stroustrup, B. [1991]. *The C++ Programming Language,* second edition, Addison-Wesley, Reading, Mass.

第 2 章

迭代、归纳和递归

计算机的威力源自其反复执行同一任务或同一任务不同版本的能力。在计算领域，*迭代*这一主题会以多种形式出现。数据模型中的很多概念（比如表）都是某种形式的重复，比如“表要么为空，要么由一个元素接一个元素，再接一个元素，如此往复而成”。使用迭代，程序和算法可以在不需要单独指定大量相似步骤的情况下，执行重复性的任务，如“执行下一步骤1000次”。编程语言使用像C语言中的`while`语句和`for`语句那样的循环结构，来实现迭代算法。

与重复密切相关的是*递归*，在递归技术中，概念是直接或间接由其自身定义的。例如，我们可以通过“表要么为空，要么是一个元素后面再跟上一个表”这样的描述来定义表。很多编程语言都支持递归。在C语言中，函数F是可以调用自身的，既可以从F的函数体中直接调用自己，也可以通过一连串的函数调用，最终间接调用F。另一个重要思想——归纳，是与“递归”密切相关的，而且常用于数学证明中。

迭代、归纳和递归都是基本概念，会以多种形式出现在数据模型、数据结构和算法中。下面介绍了一些使用这些概念的例子，每项内容都会在本书中详细介绍。

(1) *迭代技术*。反复执行一系列操作的最简单方法就是使用迭代结构，比如C语言中的`for`语句。

(2) *递归程序设计*。C语言及其他众多语言都允许函数递归，即函数可以直接或间接地调用自己。对新手程序员来说，编写迭代程序通常比写递归程序更安全，不过本书的一个重要目标就是让读者习惯在适当的时候用递归的方式来思考和编程。递归程序更易于编写、分析和理解。

符号：求和符号和求积符号

加大字号的大写希腊字母∑通常用来表示求和，如$\sum_{i=1}^{n} i$。这个特殊的表达式表示从1到n这n个整数的和，也就是$1+2+3+\cdots+n$。更加一般化的情况是，我们可以对任何具有求和指标（summation index）i的函数$f(i)$求和。（当然，这个指标也可能是i以外的一些符号。）表达式$\sum_{i=a}^{b} f(i)$就表示

$$f(a)+f(a+1)+f(a+2)+\cdots+f(b)$$

例如，$\sum_{j=2}^{m} j^2$就表示$4+9+16+\cdots+m^2$的和，这里的函数f就是“求平方”，而我们用了指标j来代替i。

作为特例，如果$b<a$，那么表达式$\sum_{i=a}^{b} f(i)$不含任何项，当然，其值也就是0了。如果$b=a$，那么表达式只有$i=a$时的那一项。因此，$\sum_{i=a}^{a} f(i)$的值就是$f(a)$。

用于求积的类似符号是个大号的大写希腊字母 Π 。表达式 $\prod_{i=a}^{b} f(i)$ 就表示

$$f(a)\times f(a+1)\times f(a+2)\times\cdots\times f(b)$$

的积，如果 $b<a$，那么该表达式的值为1。

(3) *归纳证明*。“归纳证明”是用来表明命题为真的一项重要技术。从2.3节开始，我们将广泛介绍归纳证明。下面是归纳证明最简单的一种形式。我们有与变量 n 相关的命题 $S(n)$，希望证明 $S(n)$ 为真。要证明 $S(n)$，首先要提供*依据*，也就是 n 为某个值时的命题 $S(n)$。例如，我们可以令 $n=0$，并证明命题 $S(0)$。接着，我们必须对*归纳步骤*加以证明，我们要证明，对应参数某个值的命题 S，是由对应参数前一个值的相同命题 S 得出的，也就是说，对所有的 $n\geqslant 0$，可从 $S(n)$ 得到 $S(n+1)$。例如，$S(n)$ 可能是常见的求和公式

$$\sum_{i=1}^{n} i = n(n+1)/2 \tag{2.1}$$

这是说1到 n 这 n 个整数的和等于 $n(n+1)/2$。特例可以是 $S(1)$，即等式(2.1)在 n 为1时的情况，也就是 $1=1\times 2/2$。归纳步骤就是要表明，由 $\sum_{i=1}^{n} n(n+1)/2$ 可以得出 $\sum_{i=1}^{n+1}(n+1)(n+2)/2$ ，前者就是 $S(n)$，是等式(2.1)本身，而后者则是 $S(n+1)$，就是用 $n+1$ 替换了等式(2.1)中的 n。2.3节将会为大家展示如何进行这样的证明。

(4) *程序正确性证明*。在计算机科学中，我们常希望能够证明与程序有关的命题 $S(n)$ 为真，不管是采用正式的还是非正式的方式。例如，命题 $S(n)$ 可能描述了某个循环的第 n 次迭代中什么为真，或是对某个函数的第 n 次递归调用来说什么为真。对这类命题的证明一般都使用归纳证明。

(5) *归纳定义*。计算机科学的很多重要概念，特别是那些涉及数据模型的，最好用归纳的形式来定义，也就是我们给出定义该概念最简单形式的基本规则，以及可用来从该概念较小实例构建更大实例的归纳规则。举例来说，我们提到过的表就可由基本规则（空表是表）加上归纳规则（一个元素后面跟上一个表也是表）来定义。

(6) *运行时间的分析*。算法处理不同大小的输入所花的时长（算法的“运行时间”）是衡量其“优良性”的一项重要指标。当算法涉及递归时，我们会使用名为*递推方程*的公式，它是种归纳定义，可以预测算法处理不同大小的输入所花的时间。

本章会介绍前5项主题，程序的运行时间将在第3章中介绍。

2.1　本章主要内容

在本章中，我们将介绍以下主要概念。

- 迭代程序设计（2.2节）。
- 归纳证明（2.3节和2.4节）。
- 归纳定义（2.6节）。
- 递归程序设计（2.7节和2.8节）。
- 证明程序的正确性（2.5节和2.9节）。

除此之外，通过这些概念的例子，我们还会着重介绍计算机科学中一些有趣的重要思想。其中包括：

- 排序算法，包括选择排序（2.2节）和归并排序（2.8节）。

- ❑ 奇偶校验及数据错误的检测（2.3节）。
- ❑ 算术表达式及其代数变形（2.4节和2.6节）。
- ❑ 平衡圆括号（2.6节）。

2.2　迭代

新手程序员都会学习使用迭代，采用某种循环结构（如C语言中的`for`语句和`while`语句）。在本节中，我们将展示一个迭代算法的例子——“选择排序”。在2.5节中，我们还将通过归纳法证明这种算法确实能排序，并会在3.6节中分析它的运行时间。在2.8节中，我们要展示如何利用递归设计一种更加高效的排序算法，这种算法使用了一种称作“分而治之”的技巧。

常见主题：自定义和依据–归纳

在学习本章时，大家应该注意到有两个主题贯穿多个概念。第一个是自定义（self-definition），就是指概念是依据其自身定义或构建的。例如，我们提到过，表可以定义为空，或一个元素后跟一个表。

第二个主题是依据–归纳（basis-induction）。递归函数通常都含有某种针对不需要递归调用的“依据”实例，以及需要一次或多次递归调用的“归纳”实例进行测试。众所周知，归纳证明包括依据和归纳步骤，归纳定义也一样。依据–归纳这一对非常重要，在后文中每次出现依据情况或归纳步骤时，都会突出标记这些词语。

运用恰当的自定义不会出现悖论或循环性，因为自定义的子部分总是比被定义的对象“更小”。此外，在经过有限个通向更小部分的步骤后，就能到达依据情况，也就是自定义终止的地方。例如，表L是由一个元素和比L少一个元素的表构成的。当我们遇到没有元素的表，就有了表定义的依据情况：“空表是表”。

再举个例子，如果某递归函数是有效的，那么从某种意义上讲，某一函数调用的参数必须要比调用该函数的函数副本的参数“更小”。还有，在经过若干次递归调用后，我们必须要让参数“小到”函数不再进行递归调用为止。

2.2.1　排序

要排序具有n个元素的表，我们需要重新排表中的元素，使它们按照非递减顺序排列。

✦ 示例 2.1

假设有整数表{3，1，4，1，5，9，2，6，5}。我们要将其重新排列成序列{1，1，2，3，4，5，5，6，9}，实现对该表的排序。请注意，排序不仅会整理好各值的顺序，使每个元素的值小于等于接下来那个元素的值，而且不会改变每个值出现的次数。因此，排序好的表中有两个1和两个5，而原表中只出现一次的数字都只有一个。

只要表的元素有“小于”的顺序可言，也就是具备我们通常用符号<表示的关系，就可对这些元素排序。例如，如果这些值是实数或整数，那么符号<就表示实数或整数的小于关系；如果这些值是字符串，就按字符串的词典顺序来排列（“词典顺序”的介绍详见下文附注栏）。有时候，当元素比较复杂，比如当元素是结构体时，就可能使用每个元素的一部分（比如某个特定字段）来进行比较。

词典顺序

要比较两个字符串，通常是依据它们的词典顺序进行比较的。假设$c_1c_2\cdots c_k$和$d_1d_2\cdots d_m$是两个字符串，其中每个c和每个d都只代表一个字符。字符串的长度k和m不一定是相同的。我们假设对字符而言也有<顺序，例如，在C语言中，字符就是小的整数，所以字符常量和字符变量可以在算术表达式中作为整数使用。因此，我们可以使用整数间惯有的<关系，区分两个字符中哪个字符“小于”另一个字符。这种顺序包含这样一个自然的概念，出现在字母表靠前位置的小写字母，要“小于”出现在字母表中靠后位置的小写字母，同样的道理对大写字母也成立。

这样我们可以将字符串的这种顺序称为字典、词典或字母表顺序，如下所示。如果以下任意一条成立的话，我们就说$c_1c_2\cdots c_k < d_1d_2\cdots d_m$。

(1) 第一个字符串是第二个字符串的真前缀（proper prefix），这表示$k<m$，而且对$i=1, 2, \cdots, k$而言，都有$c_i=d_i$。根据这条规则，就有`bat<batter`。作为这条规则的特例，可能有$k=0$，这样第一个字符串就不含任何字符。我们用希腊字母ϵ表示空字符串这种不含字符的字符串。当$k=0$时，规则(1)表示对任何非空字符串s而言，都有$\epsilon < s$。

(2) 对某个$i>0$的值，两个字符串的前$i-1$个字符都相同，但第一个字符串的第i个字符要小于第二个字符串的第i个字符。也就是说，对$j=1, 2, \cdots, i-1$，都有$c_j=d_j$，而且$c_i<d_i$。根据这条规则，`ball<base`，因为这两个单词是从第3个字母起开始不同的，而`ball`的第三个字母是`l`，要比`base`的第三个字母`s`更小。

$a \leqslant b$这一比较关系总是表示，要么$a<b$，要么a和b具有相同的值。如果$a_1 \leqslant a_2 \leqslant \cdots \leqslant a_n$，也就是说，如果这些值有着非递减顺序，那么我们就说表（$a_1, a_2, \cdots, a_n$）是已排序的。排序是这样一种操作，它接受任意表（$a_1, a_2, \ldots, a_n$），并生成满足如下条件的表（$b_1, b_2, \ldots, b_n$）。

(1) 表（$b_1, b_2, \cdots, b_n$）是已排序的；

(2) 表（$b_1, b_2, \cdots, b_n$）是原表的排列。也就是说，表（$a_1, a_2, \cdots, a_n$）中的每个值出现的次数，和那些值出现在（$b_1, b_2, \cdots, b_n$）中的次数是一模一样的。

排序算法接受任意的表作为输入，并生成对输入进行过排列的已排序表作为输出。

✦ 示例 2.2

考虑`base`，`ball`，`mound`，`bat`，`glove`，`batter`这列单词。有了该输入，排序算法会按照词典顺序生成输出：`ball`，`base`，`bat`，`batter`，`glove`，`mound`。

2.2.2 选择排序：一种迭代排序算法

假设要对一个具有n个整数的数组`A`按照非递减顺序排序。我们可以通过对这个步骤的迭代来完成该工作：找出尚不在数组已排序部分的一个最小元素[①]，将其交换到数组未排序部分的第一个位置。在第一次迭代中，我们在整个数组`A[0..n-1]`中找出（“选取”）一个最小元素，并将其与`A[0]`互换位置。[②]在第二次迭代中，我们从`A[1..n-1]`中找出一个最小元素，并将其与`A[1]`互换位置。继续进行这种迭代。在开始第i+1次迭代时，`A[0..i-1]`已经是将`A`中较小的i

① 这里说“一个”最小元素是因为最小值可能出现多次。如果是这样，找到任何一个最小值就行了。

② 为了描述数组中某个范围内的元素，我们采用了Pascal语言中的约定。如果A是数组，那么`A[i..j]`就表示数组A中下标从i到j这个范围内的那些元素。

个元素按照非递减顺序排序了，而数组中余下的元素则没有特定的顺序。在第i+1次迭代开始前数组A的状态如图2-1所示。

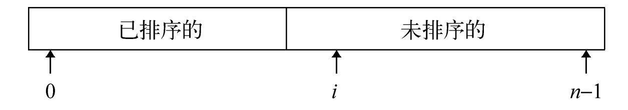


图2-1　在进行选择排序的第i+1次迭代前数组的示意图

对名字与值的约定

我们可以将变量视为具有名字和值的框。在提到变量时，比如abc，我们会使用等宽字体来表示其名字；在提到变量abc的值时，我们会使用斜体字，如*abc*。总之，abc表示框的名字，而*abc*则表示它的内容。

在第$i+1$次迭代中，要找出A[i..n-1]中的一个最小元素，并将其与A[i]互换位置。因此，在经过第$i+1$次迭代之后，A[0..i]已经是将A中较小的$i+1$个元素按照非递减顺序排序了。在经过第$n+1$次迭代之后，就完成了对整个数组的排序。

图2-2展示了用C语言编写的选择排序函数。这个名为SelectionSort的函数接受数组A作为其第一个参数。第二个参数n表示的是数组A的长度。

```
        void SelectionSort(int A[], int n)
        {
            int i, j, small, temp;
(1)         for (i = 0; i < n-1; i++) {
                /* 将 small 置为剩余最小元素第一次出现时
                   的下标*/
(2)             small = i;
(3)             for (j = i+1; j < n; j++)
(4)                 if (A[j] < A[small])
(5)                     small = j;
                /*到达这里时，small 是 A[i..n-1]
                中第一个最小元素的下标，*/
                /*现在交换 A[small]与 A[i]。 */
(6)             temp = A[small];
(7)             A[small] = A[i];
(8)             A[i] = temp;
            }
        }
```

图2-2　迭代的选择排序

第(2)到(5)这几行程序从数组未排序的部分A[i..n-1]中选取一个最小元素。我们首先在第(2)行中将下标small的值设为i。第(3)到(5)这几行的for循环会依次考量所有更高的下标j，如果A[j]的值小于A[i..j-1]这个范围内的任何数组元素的值，那么就将*small*置为j。这样一来，我们就将变量small的值置为A[i..n-1]中最先出现的那个最小元素的下标了。

在为下标small选好值后，在第(6)到(8)行中，我们要将处于该位置的元素与A[i]处的元素互换位置。如果$small=i$，交换还是会进行，只是对数组没有任何影响。请注意，要交换两个元素的位置，还需要一个临时的位置来存储二者之一。因此，我们在第(6)行将A[small]里的

值移到temp中，并在第(7)行将A[i]里的值移到A[small]中，最终在第(8)行将原来A[small]里的值从temp移到A[i]中。

✦ 示例 2.3

我们来研究一下SelectionSort针对各种输入的行为。首先看看运行SelectionSort处理没有元素的数组时会发生什么。当$n = 0$时，第(1)行中的for循环的主体不会执行，所以SelectionSort很从容地“什么事都没做”。

现在考虑一下数组只有一个元素的情况。这次第(1)行中的for循环的主体还是不会执行，这种反应是令人满意的，因为由一个元素组成的数组始终是已排序的。当n为0或1时的情况是重要的边界条件，检测这些条件下算法或程序的性能是很重要的。

最后，我们要运行SelectionSort，处理一个具有4个元素的较小数组，其中A[0]到A[3]分别是

	0	1	2	3
A	40	30	20	10

我们从i=0起开始外层的循环，并在第(2)行将small置为0。第(3)到(5)行构成了内层的循环，在该循环中，j依次被置为1、2和3。对于$j = 1$，第(4)行的条件是成立的，因为$A[1]$，也就是30，要小于$A[small]$，即$A[0]$，或者说是40。因此，在第(5)行我们会将small置为1。在(3)至(5)行第二次迭代时，有$j = 2$，第(4)行的条件还是成立，因为$A[2]<A[1]$，所以我们在第(5)行将small置为2。在第(3)到(5)行的最后一次迭代中，有$j = 3$，第(4)行的条件依旧成立，因为$A[3]<A[2]$，所以在第(5)行将small置为3。

现在我们跳出内层循环，到达第(6)行。将$A[small]$（即10）赋给temp，接着在第 (7) 行，将$A[0]$（也就是40）赋给A[3]，然后在第(8)行将10赋给A[0]。现在，外层循环的第一次迭代已经完成，而此时的数组A就变成下面这样了

	0	1	2	3
A	10	30	20	40

外层循环进行第二次迭代时，有$i = 1$，在第(2) 行将small置为1。内层循环起初会将j置为2，而因为$A[2] < A[1]$，第(5)行会将small置为2。对$j = 3$，第(4)行的条件不成立，因为$A[3] \geqslant A[2]$。因此，在到达第(6)行时，就有$small$=2。第(6)到(8)行会交换A[1]和A[2]，让数组变成

	0	1	2	3
A	10	20	30	40

虽然数组现在正好已排序了，但我们还是要迭代一次外层循环，这时$i = 2$。我们在第(2)行将small置为2，内层循环此时按照$j = 3$的情况来执行。因为第(4)行的条件不成立，$small$依旧为2，而在第(6)到(8)行中，我们会将A[2]与其自身“进行交换”。大家应该确认，当$small$=i时，这种交换是没有效果的。

键排序

在排序时，我们会对要排序的值进行比较操作。通常只对值的特定部分进行比较，而用于比较的这个部分就称为键。

例如，课表可能是具有如下形式的C语言结构体数组A

```
struct STUDENT {
    int studentID;
    char *name;
    char grade;
} A[MAX];
```

我们可能希望通过学号、学生姓名或所在年级来排序，每项内容都可以作为键。例如，如果我们希望通过学号为结构体排序，就可以在SelectionSort的第(4)行进行如下比较：

```
A[j].studentID < A[small].studentID
```

数组A和交换中使用的临时变量temp都是struct STUDENT类型，而不是integer类型的。请注意，整个结构体都要进行交换，而不仅仅是交换键字段。

交换整个结构体是很费时的，所以产生了一种更有效率的方法，即使用的另一个元素是指向STUDENT结构体的指针的数组，并且只为第二个数组中的指针排序，结构体本身在第一个数组中保持不变。我们将这种方式的选择排序留作本节的习题。

图2-3展示了SelectionSort函数如何应用到完整的程序中，来给含有n（这里约定$n \leq 100$）个整数的序列排序。第(1)行会读取并存储数组A中的n个整数。如果输入超过MAX，只有前MAX个整数被装入数组A。提供一条消息警告用户输入的数字过大在这里可能很实用，不过我们先不考虑这一点。

```
        #include <stdio.h>

        #define MAX 100
        int A[MAX];
        void SelectionSort(int A[], int n);

        main()
        {
            int i, n;
            /* 在 A 中读取和存储输入 */
(1)         for (n = 0; n < MAX && scanf("%d", &A[n]) != EOF; n++)
(2)             ;
(3)         SelectionSort(A,n); /* 排序 A */
(4)         for (i = 0; i < n; i++)
(5)             printf("%d\n", A[i]); /* 打印 A */
        }

        void SelectionSort(int A[], int n)
        {
            int i, j, small, temp;
            for (i = 0; i < n-1; i++) {
                small = i;
                for (j = i+1; j < n; j++)
                        if (A[j] < A[small])
                            small = j;
                temp = A[small];
                A[small] = A[i];
                A[i] = temp;
            }
        }
```

图2-3　使用选择排序的排序程序

第(3)行调用`SelectionSort`来为数组排序。第(4)行和第(5)行会按照排好的顺序将这些整数打印出来。

2.2.3 习题

(1) 假设用`SelectionSort`函数来处理包含如下几组元素的数组：

(a) 6，8，14，17，23

(b) 17，23，14，6，8

(c) 23，17，14，8，6

在每种情况下，分别会发生多少次元素的比较和交换？

(2) ** 在为具有n个元素的序列排序时，`SelectionSort`进行(a)比较和(b)交换的最少次数及最多次数分别是多少？

(3) 编写C语言函数，接受两个字符链表作为参数，如果第一个字符串在词典顺序上先于第二个字符串，就返回`TRUE`。提示：实现本节中描述的字符串比较算法。当两个字符串前面的字符相同时，通过在字符串尾部让该函数调用自身进行递归。除此之外，大家还可以设计迭代算法完成同样的工作。

(4) * 修改习题(3)中的程序，使其在比较过程中忽略字母的大小写。

(5) 如果所有元素都相同，选择排序会做些什么？

(6) 修改图2-3中的程序，使其在数组元素不是整数而是类型为`struct STUDENT`的结构体时执行选择排序，就像前文附注栏“键排序”中所定义的那样。假设键字段是`studentID`。

(7) * 进一步修改图2-3，使其能为任意类型T的元素排序。不过，大家可以假设某个函数*key*可以接受某个类型为T的元素作为参数，并为该元素返回某个任意类型K的键。还假设有函数*lt*接受类型为K的两个元素作为参数，且若第一个元素“小于”第二个元素，就返回`TRUE`，否则返回`FALSE`。

(8) 除了在数组`A`中使用整数下标，还可以使用指向整数的指针表示数组中的位置。使用指针重写图2-3中的选择排序算法。

(9) * 正如在前文附注栏“键排序”中提到的，如果要排序的元素是诸如类型`STUDENT`这样的大型结构体，我们可以将它们留在原数组中保持原样，并在第二个数组中对指向这些结构体的指针排序。写下选择排序的这种变形。

(10) 写一个迭代程序，打印一个整数数组中的不同元素。

(11) 使用本章开始部分所描述的符号Σ和Π来表示以下内容。

(a) 1到377中所有奇数的和。

(b) 2到n（假设n是偶数）中所有偶数的平方的和。

(c) 8到2^k中所有2的n次幂的积。

(12) 证明当small = i时，图2-2中的第(6)到(8)行（进行交换的步骤）对数组`A`没有任何影响。

2.3 归纳证明

*数学归纳法*是种实用的技巧，可用来证明命题$S(n)$对所有非负整数n都为真，或者更一般地说，对所有不小于某个下限的整数都成立。例如，在本章开头，我们提到过可以通过对n的归纳，证明对于所有的$n\geqslant 1$，命题$\sum_{i=1}^{n} i = n(n+1)/2$都为真。

现在，假设$S(n)$是有关整数n的任意命题。在对命题$S(n)$最简单的归纳证明形式中，要证明以下两个事实。

(1) *依据情况*。多为$S(0)$，不过，依据可以是对应任意整数k的$S(k)$，这样就是证明只有在$n\geqslant k$时命题$S(n)$成立。

(2) 归纳步骤。我们要证明对所有的 $n \geqslant 0$（或者如果依据为 $S(k)$，则是对所有的 $n \geqslant k$），都可由 $S(n)$ 推出 $S(n+1)$。在证明过程中的这个部分，我们假设命题 $S(n)$ 为真。$S(n)$ 称为归纳假设，而且要假设它为真，接着我们必须证明 $S(n+1)$ 为真。

命名归纳参数

我们可以通过为要证明的命题 $S(n)$ 中的变量n指定直观含义，对归纳作出解释，这种做法通常很有用。如果n如示例2-4中那样没有特殊含义，就可以说"对n进行归纳证明"。在其他例子中，n可能具有实际意义，比如示例2.6中，n表示码字中的比特数，于是就可以说，"对码字中的比特数进行归纳证明"。

图2-4展示了从0开始的归纳。对每个整数n，都有命题 $S(n)$ 要证明。对 $S(1)$ 的证明用到了 $S(0)$，对 $S(2)$ 的证明用到了 $S(1)$，以此类推，就如图中箭头所表示的。每个命题依赖前一个命题的方式是统一的。也就是说，通过对归纳步骤的一次证明，我们可以证明图2-4中箭头表示的每个步骤。

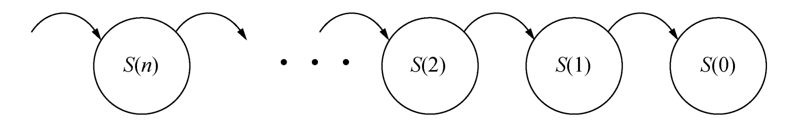


图2-4　在归纳证明中，命题$S(n)$的每个实例都是用比n的值小1的命题实例证明的

✦ 示例 2.4

作为数学归纳法的示例，我们来证明如下命题 $S(n)$

命题。对任意的 $n \geqslant 0$，都有 $S(n)$：$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

这就是说，从2的0次幂到2的n次幂，2的整数指数幂之和要比2的 $n+1$ 次幂小1①。例如，1+2+4+8 = 16−1，证明过程如下。

依据。要证明该依据，我们将等式 $S(n)$ 中的n替换为0，这样 $S(n)$ 就成了

$$\sum_{i=0}^{0} 2^i = 2^1 - 1 \tag{2.2}$$

对 $i=0$，等式(2.2)左边的和式中只有一项，这样(2.2)左边的和为2^0，也就是1。而等式(2.2)右边是 2^1-1，也就是 $2-1$，其值同样是1。因此我们证明了 $S(n)$ 的依据，也就是说，我们证明了对于 n=0，该等式成立。

归纳。现在必须要证明归纳步骤。我们假设 $S(n)$ 为真，并证明将该等式中的n替换为 $n+1$ 后等式也成立。要证明的等式 $S(n+1)$ 如下

$$\sum_{i=0}^{n+1} 2^i = 2^{n+2} - 1 \tag{2.3}$$

① 证明$S(n)$也可以不使用归纳法，只需要利用几何级数的求和公式即可。不过，该示例可以作为介绍数学归纳法的简单例子。此外，利用我们在高中可能见过的几何级数或算术级数求和公式来证明该命题是相当不严谨的，而且严格地讲，证明这些求和公式也要用到数学归纳法。

要证明等式(2.3)成立，我们先要考虑等式左侧的和

$$\sum_{i=0}^{n+1} 2^i$$

这个和几乎与 $S(n)$ 左侧的和一模一样，$S(n)$ 左侧的和为

$$\sum_{i=0}^{n} 2^i$$

只不过等式(2.3)左侧多了 $i = n+1$ 时的项，也就是 2^{n+1} 这一项。

因为可以假定归纳假设 $S(n)$ 在等式(2.3)的证明过程中为真，所以应该将 $S(n)$ 利用起来。可以将等式(2.3)中的和分为两个部分，其中之一是 $S(n)$ 中的和。也就是说，要将 $i = n+1$ 时的最后一项分离出来，将其写为

$$\sum_{i=0}^{n+1} 2^i = \sum_{i=0}^{n} 2^i + 2^{n+1} \tag{2.4}$$

现在可以利用 $S(n)$ 了，可以用 $S(n)$ 的右边 $2^{n+1}-1$ 来替换等式(2.4)中的 $\sum_{i=0}^{n} 2^i$，于是有

$$\sum_{i=0}^{n+1} 2^i = 2^{n+1} - 1 + 2^{n+1} \tag{2.5}$$

将等式(2.5)的右边简化后，它就成了 $2 \times 2^{n+1} - 1$，也就是 $2^{n+2} - 1$。现在可以看到，等式(2.5)左侧的和值，与等式(2.3)的左边相同，而等式(2.5)的右边也与等式(2.3)的右边相同。因此，就利用等式 $S(n)$ 证明了等式(2.3)的正确性，这段证明过程就是归纳步骤。由此得出的结论是，$S(n)$ 对每个非负整数n都成立。

2.3.1　归纳证明为何有效

变量的替换

在需要替换变量，比如涉及同一变量的表达式，如 $S(n)$ 中的n时，常会产生混淆。例如，我们要用 $n+1$ 替换 $S(n)$ 中的n，以得出等式(2.3)。要进行这种替换，必须先标记出S中每个出现n的地方。有个很实用的办法，就是先用某个未在S中出现过的新变量（比如m）来代替n。例如，$S(n)$ 就成了

$$\sum_{i=0}^{m} 2^i = 2^{m+1} - 1$$

接着在每个出现m的地方将其替换成所需的表达式，即本例中的n+1，就得到

$$\sum_{i=0}^{n+1} 2^i = 2^{(n+1)+1} - 1$$

若将（n+1）+1简化为$n+2$，就得到了等式(2.3)。

请注意，我们应该给用来替换的表达式加上括号，以避免意外改变运算顺序。例如，假设用 $n+1$ 替换表达式 $2 \times m$ 中的m，但没有给 $n+1$ 加上括号，那么就会得到 $2 \times n + 1$，而不是正确的表达式 $2 \times (n+1)$（也就是 $2 \times n + 2$）。

在归纳证明中，我们先证明了 $S(0)$ 为真。接下来要证明，如果 $S(n)$ 为真，那么 $S(n+1)$ 是成立的。不过为什么接着能得出 $S(n)$ 对所有 $n \geqslant 0$ 都为真呢？我们会提供两个“证据”。某位数

学家曾指出，我们证实归纳法有效的每个“证据”，都需要归纳证据本身，因此就根本没有证据。从技术上讲，归纳肯定能作为公理，然而很多人会发现以下直觉认识也是有用的。

在接下来的内容中，我们假设作为依据的值是n=0。也就是说，我们知道$S(0)$为真，而且对于所有大于0的n，如果$S(n)$为真，那么$S(n+1)$为真。如果作为依据的值是其他整数，也可以做出类似的论证。

第一个“证据”：*归纳步骤的迭代*。假设要证明对某个特定的非负整数a有$S(a)$为真。如果$a=0$，只要援引归纳依据$S(0)$的真实性即可。如果$a>0$，那么就要进行如下论证。从归纳依据可知$S(0)$为真。对于命题“$S(n)$可推出$S(n+1)$”，若将n替换为0，就成了“$S(0)$可推出$S(1)$”。因为我们知道$S(0)$为真，现在就知道$S(1)$为真。类似地，如果用1替换n，就有“$S(1)$可推出$S(2)$”，这样一来就知道$S(2)$也为真。用2来替换n，则有“$S(2)$可推出$S(3)$”，所以$S(3)$也为真，以此类推。不管a取什么值，最终都能得到$S(a)$，这样就完成了归纳。

第二个“证据”：*最少反例*。假设至少有一个n的值可以使$S(n)$不为真。设a是令$S(a)$为假的最小非负整数。如果a=0，就与我们的归纳依据$S(0)$相互矛盾，所以a一定是大于0的。不过如果$a>0$，而且a是令$S(a)$为假的最小非负整数，那么$S(a)$肯定为真。现在，在归纳步骤中，如果用$a-1$代替n，就会有$S(a-1)$可推出$S(a)$。因为$S(a-1)$为真，那么$S(a)$肯定为真，又相互矛盾了。因为我们假设存在非负整数n使$S(n)$为假，并引出了矛盾，所以$S(n)$对任何$n \geqslant 0$都一定为真。

2.3.2　检错码

现在我们要介绍“检错码”的例子。检错码本身就是个有意思的概念，而且引出了一段有趣的归纳证明。当我们通过数据网络传输信息时，会将字符（字母、数字、标点符号，等等）编码成位串，即0和1组成的序列。此时假设字符是由7位表示的。不过通常每个字符要传输不止7位，而第8位可以用来检测一些简单的传输错误。也就是说，偶尔有那么一个0或1会因为传输噪声发生改变，结果接收到的就是相反的位，进入传输线路的0成了1，而1成了0。如果通信系统能在8位中的一位发生变化时发出通知，从而发出重传信号，将会很有用。

要检测某一位的改变，必须保证任意两个表示不同字符的位序列不只有一个位置不同。不然的话，如果那个位置发生变化，结果就成了代表另一个字符的代码，可能将没法检测到错误的发生。例如，如果一个字符使用位序列01010101表示，而另一个由01000101表示，那么如果左起第4个位置发生改变，就会将前者变成后者。

要确保不同字符的代码不只有一个位置不同，方法之一是在惯用于表示字符的7位码前加上一个*奇偶校验位*。如果位序列中有奇数个1，则称其具有*奇校验*。如果位序列中有偶数个1，则其具有*偶校验*。我们选择的编码方式是以具有偶校验的8位码来表示字符，也可以选用带奇校验的代码。通过明智地选择校验位，我们可使奇偶校验成为偶校验。

✦　示例 2.5

用来表示字符`A`的传统的ASCII（音“ask-ee”，表示American Standard Code for Information Exchange，即“美国信息交换标准码”）7位码是1000001。该序列的7位中已经有偶数个1，所以我们为其加上前缀0，得到01000001。用来表示`C`的传统代码是1000011，这和表示`A`的7位码只在第6位是不同的。不过，这个代码具有奇校验，所以我们给它加上前缀1，从而产生具有偶校验的8位码11000011。请注意，在给表示`A`和`C`的代码前加上校验位后，就有了01000001和11000011这两个位序列，它们的第1位和第7位这两位是不同的，如图2-5所示。

A:	0	1	0	0	0	0	0	1
C:	1	1	0	0	0	0	1	1

图2-5 可以选择初始奇偶校验位，使得8位码总是具有偶校验

我们总是能选择一个奇偶校验位加到7位码上，从而让得到的8位码中有偶数个1。如果表示字符的7位码本来就有偶校验，就选择0作为其奇偶校验位，而对本具有奇校验的7位码，则是选择奇偶校验位1。不管哪种情况，8位码中都是包含偶数个1。

两个具有偶校验的位序列不可能只有一个位置不同。如果两个这样的位序列只有一个位置不同，那么其中一个肯定要比另一个多一个1。因此，一个序列必然具有奇校验，而另一个则是偶校验，与我们都具有偶校验的假设是矛盾的。因此可得，通过加上奇偶校验位使1的数量为偶，可为字符创建检错码。

奇偶校验位模式是相当“高效”的，从某种意义上讲，它让我们可以传输很多不同的字符。请注意，n位的位序列有2^n个，因为我们可以为第一位选择二值（0或1）之一，可以为第二位选择二值之一，等等，总共可形成$2\times2\times\cdots\times2$（$n$个2相乘）个位串，所以，最多有望能用8位来表示$2^8=256$个字符。

然而，在奇偶校验模式中，只能选择其中7位，第8位是无从选择的。因此最多可以表示2^7，即128个字符，而且能检测某一位上的错误。这样也不错，我们可以用256个中的128个，也就是8位码所有可能组合的一半，来作为字符的合法代码，还能检测某一位中出现的错误。

类似地，如果我们使用n位的位序列，选择其中一位作为奇偶校验位，那么就能用$n-1$位的位序列加上合适的奇偶校验位前缀（其值由另外那$n-1$位确定）来表示2^{n-1}个字符。n位的位序列有2^n个，我们可以表示2^n中的2^{n-1}个，或者说是可能字符数的一半，而且可以检测位序列中任意一位的错误。

有没有可能检测多个错误，并使用超过位序列多于一半的可能组合作为合法代码呢？下一个例子将告诉你这是不可能的。这里的归纳证明使用的命题对0来说不为真，所以我们必须选用一个更大的归纳依据，也就是1。

✦ 示例 2.6

我们要对n进行归纳，以证明以下命题。

命题 $S(n)$。如果C是长度为n的位串的检错（即两个不同的位串不会刚好只有一个位置不同）集合，那么C最多含有2^{n-1}个位串。

这个命题对$n=0$来说不为真。$S(0)$表示长度为0的位串的检错集合最多只有2^{-1}个，也就是半个位串。从技术上讲，只由空位串（不含任何位置的位串）组成的集合C，是长度为0的位串的检错集合，因为C中任意两个位串不会只有一个位置不同。集合C中不只是有半个位串，它其实有一个位串。因此，$S(0)$为假。不过，对于所有的$n\geqslant1$，$S(n)$都为真，正如我们在下文将会看到的。

依据。依据为$S(1)$，也就是，任何检错的长度为1的位串的集合最多只有$2^{1-1}=2^0=1$个位串。长度为1的位串只有两个，一个是位串0，一个是位串1。然而，在检错的集合中，我们不能同时拥有这两者，因为它们正好只有一个位置不同。因此，每个$n=1$的检错集合肯定最多只有一个位串。

归纳。设$n\geqslant1$，假定归纳假设——长度为n的位串的检错集合最多只有2^{n-1}个位串——为真。

我们必须用这一假设证明，任何长度为 $n+1$ 的位串的检错集合C最多只有 2^n 个位串。因此，将C分为两个集合：C_0，即C中0开头的位串组成的集合，以及C_1，即C中1开头的位串组成的集合。例如，假设 $n=2$，C就是长度为 $n+1=3$，而且有一个奇偶校验位的位串的集合。那么，如图2-6所示，C由位串000、101、110和011组成，C_0由位串000和011组成，C_1则由位串101和110组成。

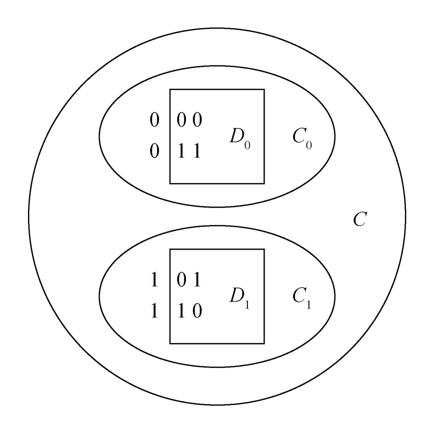


图2-6　集合C被分为0开头位串的集合C_0和1开头位串的集合C_1，D_0和D_1则分别由删除了开头的0和1的位串组成

考虑一下集合D_0，它含有删除了C_0中那些位串开头的0后形成的位串。在上面的例子中，D_0含有位串00和11。我们要求D_0不能含有两个只有一位不同的位串。原因在于，如果有这样两个位串，比方说$a_1a_2\cdots a_n$和$b_1b_2\cdots b_n$，然后恢复它们开头的0，就会给出两个C_0中的位串，$0a_1a_2\cdots a_n$和$0b_1b_2\cdots b_n$，而这两个位串也只有一位是不同的。不过C_0中的位串也是C中的元素，而且我们知道C中不能有两个位串只有一个位置不同。因此，D_0也不行，所以D_0是检错集合。

现在可以应用该归纳假设得出，D_0作为一个长度为n的位串的检错集合，最多有 2^{n-1} 个位串。因此，C_0最多有 2^{n-1} 个位串。

同样，可以对C_1集合作出类似推论。设D_1集合内的元素是删除C_1中位串开头的1形成的位串。D_1是长度为n的位串的检错集合，而根据归纳假设，D_1最多只有 2^{n-1} 个位串。因此，C_1也最多只有 2^{n-1} 个位串。然而，C中的每个位串不是在C_0中就是在C_1中。因此，C中最多有 $2^{n-1}+2^{n-1}$ 个，也就是2^n个位串。

我们已经证明了 $S(n)$ 可推出 $S(n+1)$，所以可以得出结论，$S(n)$ 对所有的 $n\geqslant 1$ 都为真。我们在声明中排除了 $n=0$ 的情况，因为归纳依据是 $n=1$，而不是 $n=0$。现在看到带奇偶校验检查的检错集合是尽可能大的，因为它们在使用n个位来构成位串时能有 2^{n-1} 个位串。

如何构造归纳证明

没有什么可以保证给出任意（真）命题 $S(n)$ 的归纳证明。找到归纳证明，就像找到任意类型的证明那样，或者就像写出能正常运行的程序那样，是项挑战智力的任务，而且我们只有几句话的意见可提。如果大家研究了示例2.4和示例2.6中的归纳步骤，就会注意到，每种情况下，都必须对试图证明的命题 $S(n+1)$ 加以处理，使其由归纳假设 $S(n)$ 和某些额外内容组成。在示例2.4中，我们将和

$$1+2+4+\cdots+2^n+2^{n+1}$$

表示为归纳假设告诉我们的和

$$1+2+4+\cdots+2^n$$

加上2^{n+1}这一项。

在示例2.6中，我们用两个长度为n的位串集合（称为D_0和D_1）表示长度为$n+1$的位串集合C，这样一来，就可以将归纳假设应用到这些集合上，并推断出这两个集合都是大小有限的。

当然，对命题$S(n+1)$加以处理，从而使我们能应用归纳假设，只是更普遍的解题箴言“运用给定条件”的一个特例。当必须处理$S(n+1)$的“额外”部分，并利用$S(n)$完成对$S(n+1)$的证明时，才是最让人头疼的。不过，以下规则是普遍适用的。

- 归纳证明必须在某个地方表述“……而且通过归纳假设我们可知……”。如果没有的话，就不算是归纳证明。

2.3.3 习题

(1) 通过对n从$n=1$起进行归纳，证明以下公式。

(a) $\sum_{i=1}^{n} i = n(n+1)/2$

(b) $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$

(c) $\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4$

(d) $\sum_{i=1}^{n} 1/i(i+1) = n/(n+1)$

(2) 形如$t_n = n(n+1)/2$的数字称为*三角形数*，因为将弹珠排列成等边三角形，每条边上排n个，那么弹珠的总数就是$\sum_{i=1}^{n} i$，而从我们在习题(1)中证明的结论可知这是t_n个弹珠。例如，保龄球瓶排列成每条边上有4个球瓶的等边三角形，共有$t_4 = 4\times 5/2 = 10$个保龄球瓶。用归纳法证明$\sum_{i=1}^{n} t_j = n(n+1)(n+2)/6$。

(3) 判断以下位序列的奇偶校验是偶校验还是奇校验。

(a) 01101
(b) 111000111
(c) 010101

(4) 假设我们用3个数字，比如0、1和2，来为符号编码。由0、1和2组成的位串集合C中，如果任意两个位串不只有一个位置不同，那么这个集合就是检错的。例如，{00，11，22}就是长度为2的位串的检错集合。证明对任意的$n\geqslant 1$，使用数字0、1和2组成的长度为n的位串的检错集合最多只有3^{n-1}个位串。

(5) * 证明：对任意的$n\geqslant 1$，存在使用0、1和2三个数字组成的长度为n的位串的检错集合，其中含有3^{n-1}个位串。

(6) * 证明：如果使用k个符号，对任意的$k\geqslant 2$，都有使用k个不同符号作为“数字”并且长度为n的位串的检错集合，其中具有k^{n-1}个位串，但这样的位串集合肯定不可能含有超过k^{n-1}个位串。

(7) * 如果$n\geqslant 1$，则使用0、1和2这三个数字组成的位串中，连续位置完全不具相同数字的位串共有$3\times 2^{n-1}$个。例如，长度为3的此类位串共有：010、012、020、021、101、102、120、121、201、202、210和212。通过对位串的长度进行归纳来证明该结论。这个公式对n=0来说是否为真？

(8) * 证明：1.3节中讨论过的行波进位加法算法能产生正确的答案。*提示*：通过对i的归纳证明，考虑从右端起的i位，两个加数后i位的和，其二进制形式为进位位后跟上目前为止所生成的i位结果。

(9) * 含n个项的几何级数$a, ar^2, ar^3, \cdots, ar^{n-1}$的和公式是

$$\sum_{i=0}^{n-1} ar^i = \frac{(ar^n - a)}{(r-1)}$$

通过对n的归纳来证明该公式。请注意，要让公式成立，必须假设$r \neq 1$。在证明过程中会在哪里用到这一假设呢？

(10) 第一项为a，公差为b的算术级数$a, (a+b), (a+2b), \cdots, (a+(n-1)b)$的求和公式为

$$\sum_{i=0}^{n-1} a + bi = n\big(2a + (n-1)b\big) / 2$$

(a) 通过对n的归纳证明该公式。

(b) 证明习题(1)中的(a)也是该公式的一例。

(11) 给出两段非正式的证明，表明虽然命题$S(0)$为假，但归纳可以从1开始“起效”。

(12) 通过对位串长度的归纳证明，由奇校验位串构成的代码也可以检错。

(13) ** 如果某种编码中任意两个位串不同的位置不少于3位，那么我们就可以通过找出该编码中与接收到的位串仅有一位不同的唯一位串，纠正单个错误。事实证明，有一种针对7位位串的编码，它可以纠正单个错误并含有16个位串。试着找出这种编码。提示：推理出来可能是最佳方法，不过如果推理失败，可以写程序来搜索这样的编码。

(14) * 偶校验码可否检出“双重错误”，也就是两个不同位上的改变？它能否纠正单个错误？

算术和与几何和

高中代数中的两个公式我们会经常用到。它们都有着有趣的归纳证明，也就是我们在习题(9)和习题(10)中让读者证明的。

算术级数（即等差数列）是一列具有以下形式的n个数字。

$$a, (a+b), (a+2b), \cdots, (a+(n-1)b)$$

第一项为a，而每一项都要比前一项大b。这n个数字的和，就是第一项和最后一项的平均数的n倍，也就是

$$\sum_{i=0}^{n-1} a + bi = n(2a + (n-1)b) / 2$$

例如，考虑一下$3+5+7+9+11$的和。总共有$n=5$项，第一项为3，最后一项为11。因此，这个和就是$5 \times (3+11)/2 = 5 \times 7 = 35$。可以把这5个数加起来，来证明这个和是正确的。

几何级数（即等比数列）是一列具有如下形式的n个数字。

$a, ar, ar^2, ar^3, \cdots, ar^{n-1}$

也就是说，第一项为a，而每一项都是前一项的r倍。n项几何级数的和公式是

$$\sum_{i=0}^{n-1} ar^i = \frac{(ar^n - a)}{(r-1)}$$

在这里，r可以大于1，也可以小于1。如果$r=1$的话，以上公式就不可用了，不过所有项都是a，其和也很明显，就是an。

作为几何级数求和的例子，考虑一下1+2+4+8+16。这时$n=5$，第一项a就是1，而公比$r=2$，因此这个和就是

$$(1 \times 2^5 - 1)/(2-1) = (32-1)/1 = 31$$

再举一个大家可以验证的例子，考虑$1+1/2+1/4+1/8+1/16$。还是$n=5$而且$a=1$，不过$r=1/2$，这个和就是

$$(1 \times (\frac{1}{2})^5 - 1)/(\frac{1}{2} - 1) = (-31/32)/(-1/2) = 1\frac{15}{16}$$

简单归纳的模板

我们对2.3节进行总结，给出适用于该节中归纳证明过程的简单模板。2.4节中将介绍更为通用的模板。

(1) 指定待证明的命题 $S(n)$ 。表明自己要通过对n的归纳，对所有 $n \geqslant i_0$ ，证明 $S(n)$ 。这里的i0是作为归纳依据的常数，通常i0是0或1，不过它也可以是任意整数。直观地解释n的含义，比如，n是码字的长度。

(2) 陈述依据情况， $S(i_0)$ 。

(3) 证明依据情况，也就是解释 $S(i_0)$ 为何为真。

(4) 陈述对某些 $n \geqslant i_0$ ，假设有 $S(n)$ ，也就是陈述“归纳假设”，建立归纳步骤。用$n+1$替换命题 $S(n)$ 中的n来表示 $S(n+1)$ 。

(5) 假定归纳假设 $S(n)$ 为真，证明 $S(n+1)$ 。

(6) 得出 $S(n)$ 对所有 $n \geqslant i_0$ 都（但对更小的n不一定）为真结论。

2.4　完全归纳

目前为止所看到的例子，在证明 $S(n+1)$ 为真时，都只用到了 $S(n)$ 作为归纳假设。不过，由于要对参数从归纳依据开始增加的值证明命题S，我们可以对从归纳依据到n的所有i的值使用 $S(i)$ ，这种形式的归纳叫作*完全归纳*（有时也称为*完美归纳*或*强归纳*）。而2.3节所示的简单归纳形式，也就是只用 $S(n)$ 来证明 $S(n+1)$ ，有时被称为*弱归纳*。

先来考虑一下如何进行从归纳依据 $n=0$ 开始的完全归纳。要通过以下两个步骤来证明 $S(n)$ 对所有 $n \geqslant 0$ 为真。

(1) 先证明归纳依据， $S(0)$ 。

(2) 假设 $S(0)$, $S(1)$, …, $S(n)$ 全为真，作为归纳假设。从这些命题来证明 $S(n+1)$ 成立。

至于在2.3节中描述的弱归纳，也可以在选择0之外再选择某个值a作为归纳依据，然后证明 $S(a)$ 归纳依据。而且在归纳步骤中，可以只假定 $S(a)$，$S(a+1)$, … $S(n)$ 为真。请注意，弱归纳是完全归纳的一个特例，应用弱归纳，我们在之前的命题中只选择 $S(n)$ 来证明 $S(n+1)$ 。

图2-7表示了完全归纳的原理。命题 $S(n)$ 的每个实例在其证明过程中都可以使用下标比其小的任意实例。

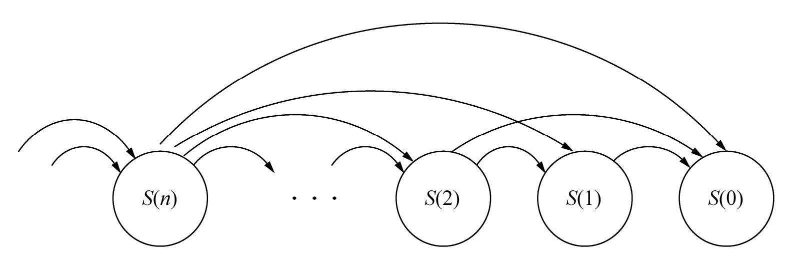


图2-7　完全归纳允许每个实例在其证明过程中使用在它之前的一个、一些或是所有实例

2.4.1　使用多个依据情况进行归纳

在进行完全归纳时，拥有多个依据情况往往是很实用的。如果希望证明命题 $S(n)$ 对所有 $n \geqslant i_0$ 都为真，那么不仅可以用i_0作为依据情况，而且能用一些大于 i_0 的连续整数（假设是 i_0，i_0+1, i_0+2, …, j_0 ）作为依据情况。然后我们必须完成以下两步。

(1) 证明每个依据情况，即命题$S(i_0)$，$S(i_0+1)$,⋯，$S(j_0)$。

(2) 假设对于某个$n \geqslant j_0$，$S(i_0)$，$S(i_0+1)$，⋯，$S(n)$全成立，作为归纳假设，并证明$S(n+1)$为真。

✦ 示例 2.7

第一个完全归纳的例子是使用多个依据情况的简单例子。正如我们将要看到的，它只是有限程度的“完全”。为了证明$S(n+1)$，我们没有使用$S(n)$，而只使用了$S(n-1)$。在更普遍的完全归纳推理中，我们要使用$S(n)$、$S(n-1)$以及命题S的很多其他实例。

下面通过对n的归纳来对所有的$n \geqslant 0$证明以下命题。[1]

命题 $S(n)$。总是存在整数a和b（正整数、负整数或0），使$n = 2a + 3b$。

依据。我们同时采用0和1作为依据情况。

(1) 对于$n = 0$，可以选用$a = 0$和$b = 0$。显然$0 = 2\times 0 + 3\times 0$。

(2) 对于$n = 1$，可以选用$a = -1$和$b = 1$。然后有$1 = 2\times(-1) + 3\times 1$。

归纳。现在，可对任意的$n \geqslant 0$，假设$S(n)$为真，并证明$S(n+1)$为真。请注意，可假设n至少是从我们已证明的依据（这里$n \geqslant 1$）起的连续值中最大的那个。而命题$S(n+1)$就是说存在某些整数a和b，使得$n+1 = 2a + 3b$。

归纳假设表明$S(0)$，$S(1)$,⋯，$S(n)$全部为真。请注意，序列从0开始是因为它是连续依据情况的下限。因为可以假设$n \geqslant 1$，我们知道$n-1 \geqslant 1$，因此$S(n-1)$为真。该命题就是说，存在整数a和b，使得$n+1 = 2a + 3b$。

由于命题$S(n+1)$中需要用到a，因此这里重新声明$S(n-1)$使用不同名称的整数，比方说存在整数a'和b'，使得

$$n - 1 = 2a' + 3b' \tag{2.6}$$

如果给(2.6)的两边都加上2，就得到$n+1 = 1(a'+1) + 3b'$。如果接着令$a = a'+1$，$b = b'$，那么就存在整数a和b，使得命题$n+1 = 2a + 3b$为真。该命题就是$S(n+1)$，所以我们已经证明了该归纳推理。请注意，在证明过程中，没有用到$S(n)$，但用到了$S(n-1)$。

2.4.2 验证完全归纳

就像2.3节中讨论的普通归纳（或“弱”归纳）那样，通过“最少反例”论证，完全归纳也可以被直观地证实为一种证明技巧。令依据情况为$S(i_0)$，$S(i_0+1)$，⋯，$S(j_0)$，并假设已经证明了对任意的$n \geqslant j_0$，$S(i_0)$，$S(i_0+1)$，⋯，$S(n)$能一起推出$S(n+1)$。现在，假设至少存在一个不小于i_0的n值使$S(n)$不成立，并设b是令$S(b)$为假的最小的不小于i_0的整数。那么b就不能是i_0和j_0之间的整数，否则与归纳依据矛盾。此外，b也不能大于j_0。不然，$S(i_0)$，$S(i_0+1)$，⋯，$S(b-1)$全为真。而归纳步骤接着就会告诉我们$S(b)$也为真，这样就产生了矛盾。

2.4.3 算术表达式的规范形式

现在探讨将算术表达式变形为等价形式的例子。它表明完全归纳利用了可假设待证明的命题S对所有n以下（包含n）的参数都为真这一事实。

---

① 其实，这个命题对所有的n，不论n是正整数还是负整数，都是成立的，不过n为负整数的情况需要另外进行归纳推理，我们将这个证明过程留给大家作为习题。

作为一种激励形式，编程语言的编译器可以利用算术运算符的代数形式，重新排列所计算的算术表达式中操作数的顺序。这种重排的目标是为计算机找出一种比表达式原有计算顺序耗时更少的方式来计算该表达式。

在本节中，只考虑含有一种结合和交换运算符（比如+）的算术表达式，并看看可以对操作数进行怎样的重新排列。我们将证明，如果有任意只含“+”运算符的表达式，那么该表达式的值，要与其他任何只对同样操作数使用“+”的表达式的值相等，不管以何种顺序排列及（或）以何种形式组合。例如

$$\big(a_3+(a_4+a_1)\big)+(a_2+a_5)=a_1+\Big(a_2+\big(a_3+(a_4+a_5)\big)\Big)$$

我们将进行两段单独的归纳推理，以证明这一说法，其中第一段归纳推理是完全归纳。

结合性和交换性

回想一下加法结合律，就是说在求三个数的和时，既可以将前两个数相加，然后加上第三个数得到结果，也可以用第一个数，加上第二个数与第三个数相加的结果，两种情况下结果是相同的。形如：

$$(E_1+E_2)+E_3=E_1+(E_2+E_3)$$

其中，E_1、E_2和E_3都是算术表达式。例如，

$$(1+2)+3=1+(2+3)$$

这里有E_1=1、E_2=2，以及E_3=3。还比如

$$\big((xy)+(3z-2)\big)+(y+z)=xy+\big((3z-2)+(y+z)\big)$$

这里有E_1=xy，E_2=3z–2，以及E_3=y+z。

接着回想一下加法交换律，就是说可以将两个表达式按照任意顺序相加。形如：

$$E_1+E_2=E_2+E_1$$

例如，$1+2=2+1$，以及$xy+(3z-2)=(3z-2)+xy$。

✦ 示例 2.8

我们要对n（表达式中操作数的数目）进行完全归纳，以证明命题$S(n)$成立。

命题$S(n)$。如果E是含有“+”运算符和n个操作数的表达式，而a是其中一个操作数，那么可以通过使用结合律和交换律，将E变形成$a+F$的形式，其中表达式F含有E中除a之外的所有操作数，而且这些操作数是使用“+”运算符以某种顺序组合在一起的。

命题$S(n)$只对$n\geqslant 2$成立，因为表达式E中至少要出现一次“+”运算符。因此，我们要使用$n=2$作为归纳依据。

依据。令$n=2$。那么E只可能是$a+b$或$b+a$，如果说a之外的那个操作数是b的话。在$a+b$中，令F为表达式b，那么命题就成立了。而在$b+a$的情况下，注意到通过使用加法交换律，$b+a$可以变形为$a+b$，因此我们就可以再次令$F=b$。

归纳。设E有$n+1$个操作数，并假设$S(i)$对$i=2, 3, \cdots, n$都为真。我们需要为$n\geqslant 2$证明该归纳步骤，所以可假设E最少有3个操作数，也就是至少出现两次“+”运算符。可以将E写为E_1+E_2，其中E_1和E_2是某些表达式。因为E中正好有$n+1$个操作数，而且E_1和E_2都一定至少含有这些操作

数中的一个，这样一来E_1和E_2中的操作数都不能超过n个。因此，归纳假设适用于E_1和E_2，只要它们都不止有一个操作数（因为我们开始时将$n=2$作为依据）。有4种情况必须考虑：a是在E_1中还是在E_2中，以及a是否为E_1或E_2中唯一的操作数。

(a) E_1就是a本身。当E为$a+(b+c)$时，就是这种情况。这里E_1就是a，而E_2就是$b+c$。在这种情况下，E_2就是F，也就是说，E本身就已经是$a+F$的形式。

(b) E_1含有多个操作数，a是其中一个。比如

$$E=(c+(d+a))+(b+e)$$

其中$E_1=c+(d+a)$，$E_2=b+e$。这里，因为E_1的操作数不超过n个，但至少达到了两个，所以可以应用归纳假设，使用交换律和结合律，将E_1变形为$a+E_3$。因此，E可以变形为$(a+E_3)+E_2$。对该式应用结合律，就能将E进一步变形为$a+(E_3+E_2)$。这样，我们就可以选择F为E_3+E_2，这就证明了这种情况下的归纳步骤。对本例中的E，也可以假设将$E_1=c+(d+a)$变形为$a+(c+d)$。那么E就可以重新分组为$a+((c+d)+(b+e))$。

(c) E_2就是a。例如，$E=b+(a+c)$。这种情况下，可以用交换律将E变形为$a+E_1$，如果令F为E_1，这就是我们想要的形式。

(d) E_2含有包括a在内的多个操作数。比方说，$E=\ b+(a+c)$，这时可以用交换律将E变形成E_2+E_1，这样就成了情况(b)。如果$E=\ b+(a+c)$，可将E先变形为$(a+c)+b$。通过归纳假设，$a+c$可以转换成所需的形式，事实上，结果已经出来了。然后结合律就将E变形为$a+(c+b)$。

在这4种情况中，都是将E变形为所需的形式。因此，归纳步骤得到了证明，可以得出$S(n)$对所有的$n\geqslant 2$都为真的结论。

✦　示例 2.9

示例2.8中的归纳证明直接引出了一种将表达式转换成所需形式的算法。考虑如下表达式作为例子：

$$E=(x+(z+v))+(w+y)$$

假设v是我们希望“拉出来”的那个操作数，也就是扮演示例2.8的变形中a的那个角色。一开始，我们介绍一个符合情况(b)的例子，其中$E_1=x+(z+v)$，而$E_2=w+y$。

接着，必须对表达式E_1进行处理，从而将v“拉出来”。E_1符合情况(d)，因此我们先用交换律将其变形为$(z+v)+x$。作为情况(b)的实例，必须对表达式$z+v$（情况(c)的实例）加以处理，因此要通过交换律将其变形为$v+z$。

现在E_1被变形为$(v+z)+x$，接着使用结合律将其变形成$v+(z+x)$，也就是将E变形成了$(v+(z+x))+(w+y)$。通过结合律，可把E变形为$(v+(z+x))+(w+y)$。因此，$E=v+F$，其中F就是表达式$(z+x)+(w+y)$。图2-8总结了整个变形过程。

$$\begin{array}{c}
\big(x+(z+v)\big)+(w+y)\\
\big((z+v)+x\big)+(w+y)\\
\big((v+z)+x\big)+(w+y)\\
\big(v+(z+x)\big)+(w+y)\\
v+\big((z+x)+(w+y)\big)
\end{array}$$

图2-8　使用交换律和结合律，可以将任意操作数（比如v）“拉出来”

现在，可以使用示例2.8中证明过的命题来证明我们的原始论点，也就是说任意两个只涉及+运算符与同一些不同操作数的表达式，都可以通过结合律和交换律相互变形。这里是用2.3节中讨论的弱归纳证明的，没有使用完全归纳。

✦ 示例 2.10

让我们通过对表达式中操作数的个数n的归纳证明以下命题。

命题$T(n)$。如果E和F是只含+运算符以及同一组n个不同操作数的表达式，那么可以通过多次应用结合律和交换律将E变形为F。

依据。如果n=1，那么两个表达式肯定都只有一个操作数a。因为E和F是相同的表达式，所以E确实“可变形为”F。

归纳。假设$T(n)$对某些$n \geqslant 1$为真，现在要证明$T(n+1)$为真。设E和F是具有同一组$n+1$个操作数的表达式，由于$n+1 \geqslant 2$，那么示例2.8中的命题$S(n+1)$必然成立。因此，我们可将E变形为$a+E_1$，其中E_1是含有E中其他n个操作数的表达式。类似地，可以将F变形为$a+F_1$，其中F_1与E_1含有相同的n个操作数。更重要的是，在这种情况下，我们还可以进行逆向的变形，使用结合律和交换律将$a+F_1$变形为F。

现在可以对E_1和F_1援引归纳假设$T(n)$。这两个表达式具有相同的n个操作数，因此归纳假设可以应用。这就是说我们可将E_1变形为F_1，所以可将$a+E_1$变形为$a+F_1$。因此我们可以通过如下变形

$$
\begin{aligned}
E \to \cdots \to & a + E_1 && \text{使用 } S(n) \\
\to \cdots \to & a + F_1 && \text{使用 } T(n) \\
\to \cdots \to & F && \text{逆向使用 } S(n+1)
\end{aligned}
$$

将E变形为F。

✦ 示例 2.11

让我们将$E=(x+y)+(w+z)$变形为$F=((w+z)+y)+x$。先选择一个要“拉出来”的操作数，比如说是w。如果审视示例2.8中的情况，就会发现我们对E进行了一系列变形

$$(x+y)+(w+z) \to (w+z)+(x+y) \to w+\big(z+(x+y)\big) \tag{2.7}$$

而对F进行了如下变形

$$\big((w+z)+y\big)+x \to \big(w+(z+y)\big)+x \to w+\big((z+y)+x\big) \tag{2.8}$$

现在有了将$z+(x+y)$变形为$(z+y)+x$的子问题。我们要通过将x“拉出来”来解决这一问题，需要进行的变形是

$$z+(x+y) \to (x+y)+z \to x+(y+z) \tag{2.9}$$

和

$$(z+y)+x \to x+(z+y) \tag{2.10}$$

这又带来了将$y+z$变形为$z+y$的子问题，只要应用交换律便可解决该问题。严格地说，我们使用了示例2.8的技术，“拉出”了每个表达式中的y，为每个表达式留下$y+z$。然后示例2.10中的依据情况告诉我们，表达式z可以“变形为”它本身。

通过行(2.9)中的步骤，可以将$z+(x+y)$变形为$(z+y)+x$，接着对子表达式$y+z$应用交换律，最后再反向使用行(2.10)中的变形。我们把这些变形当作将$(x+y)+(w+z)$变形为$((w+z)+y)+x$的中间过程。首先要应用行(2.7)中的变形，接着用刚讨论的变形将$z+(x+y)$变

形为 $(z+y)+x$ ，最后再反向使用行(2.8)中的变形。整个变形过程可概括为图2-9所示的情况。

$(x \quad y) \quad (w \quad z)$	表达式E
$(w+z)+(x+y)$	(2.7)的中间形式
$w+\big(z+(x+y)\big)$	(2.7)的最终形式
$w+\big((x+y)+z\big)$	(2.9)的中间形式
$w+\big(x+(y+z)\big)$	(2.9)的最终形式
$w+\big(x+(z+y)\big)$	交换律
$w+\big((z+y)+x\big)$	反向使用(2.10)
$\big(w+(z+y)\big)+x$	反向使用(2.8)的中间形式
$\big((w+z)+y\big)+x$	表达式F，反向使用(2.8)的最终形式

图2-9　使用交换律和结合律将一个表达式变形为另一个表达式

2.4.4　习题

所有归纳推理的模板

以下形式的归纳证明，涵盖了具有多个依据情况的完全归纳。它还将2.3节中介绍的弱归纳作为一种特例包含其中，并包含了只有一个依据情况的一般情况。

(1) 指定要证明的命题 $S(n)$ 。声明要通过对n的归纳，证明 $S(n)$ 对 $n \geqslant i_0$ 为真。指定 i_0 的值，通常是0或1，但也可以是其他整数。直观地解释n表示什么。

(2) 陈述依据情况（一个或多个）。这些将是从 i_0 起到某个整数 j_0 的所有整数。通常 $j_0 = i_0$ ，不过 j_0 也可以是其他整数。

(3) 证明各个依据情况 $S(i_0)$, $S(i_0+1)$, …, $S(j_0)$ 。

(4) 声明假设 $S(i_0)$, $S(i_0+1)$, …, $S(n)$ 为真（就是“归纳假设”），并要证明 $S(n+1)$ ，以此来建立归纳步骤。声明自己在假设 $n \geqslant j_0$ ，也就是n至少要跟最大的依据情况一样大。通过用 $n+1$ 替换 $S(n)$ 中的n来表示 $S(n+1)$ 。

(5) 在(4)中提到的假设下证明 $S(n+1)$ 。如果归纳为弱归纳而不是完全归纳，那么证明中只需要用到 $S(n)$ ，不过用归纳假设中的任一或全部命题都是可以的。

(6) 得出 $S(n)$ 对所有的 $n \geqslant i_0$ （但不一定对更小的n）都为真。

(1) 从表达式 $E=(u+v)+\big((w+(x+y))+z\big)$ 中依次“拉出”每个操作数。也就是说，从E的每个部分开始，并使用示例2.8中的技巧将E变形为 $u+E_1$ 这样的表达式。接着再将 E_1 变形为 $v+E_2$ 这样的表达式，以此类推。

(2) 使用示例2.10中的技巧完成以下变形。

(a) 将 $w+\big(x+(y+z)\big)$ 变形为 $\big((w+x)+y\big)+z$ 。

(b) 将 $(v+w)+\big((x+y)+z\big)$ 变形为 $\big((y+w)+(v+z)\big)+x$

(3) * 设E是含+、−、∗和 / 这几种运算符的表达式，其中每种运算符都是二元的，也就是说，这些运算符都接受两个操作数。对运算符在E中出现的次数进行完全归纳，证明如果E中出现n个运算符，那么E具有 $n+1$ 个操作数。

(4) 给出一个具有交换性但不具结合性的二元运算符。

(5) 给出一个具有结合性但不具交换性的二元运算符。

(6) * 考虑运算符全为二元运算符的表达式E。E的长度是指E中符号的数目，将一个运算符或左边括号或右边括号记作一个符号，并将任一操作数（比如123或`abc`）记作一个符号。证明E的长度肯定为奇数。提示：通过对表达式E的长度进行完全归纳来证明该声明。

(7) 证明：每个负整数都可以写成 $2a+3b$ 的形式，其中a和b都是整数（不一定是正整数）。

(8) * 证明：每个整数（正整数或负整数）都可以写为 $5a+7b$ 的形式，其中a和b都是整数（不一定是正整数）。

(9) * 弱归纳证明（如2.3节中那些）是否也是完全归纳证明？完全归纳证明是否也是弱归纳证明？

(10) * 在本节中我们展示了如何通过最少反例论证来验证完全归纳。这表明了完全归纳也可通过迭代来验证。

真相大揭露

在证明程序正确的过程中，存在很多理论上和实践上的困难。一个很明显的问题是：“程序‘正确’表示什么意思？”正如我们在第1章中提到过的，多数在练习中编写的程序只满足某些非正式的规范，这些规范本身可能是不完整或不一致的。即便是存在确切的正式规范，我们也可以证明，并不存在可以证明任意的程序等同于给定规范的某个算法。

尽管存在这些困难，但陈述并证明与程序有关的断言还是有好处的。程序的循环不变式（loop invariant）通常是人们可以给出的最实用的程序工作原理的简短解释。此外，程序员在编写一段代码时，应该将循环不变式谨记心头。也就是说，程序能正常工作一定是存在某些原因的，而这个原因通常必须与程序每次进行循环或每次执行递归调用时都成立的归纳假设相关。程序员应该能设想出一个证明过程，即使行逐行把证明过程写下来可能并不现实。

2.5 证明程序的属性

在本节中，我们将深入到这样一个领域：证明程序能完成它声称能做的工作。在这个领域中，归纳证明起着举足轻重的作用。我们将看到一项技术，它可以解释迭代程序在进行循环的过程中在做些什么。如果理解循环在做什么，基本上就能明白需要对迭代程序有哪些了解。在2.9节中，我们会介绍证明递归程序的属性需要些什么。

2.5.1 循环不变式

要证明程序中循环的属性，关键是要选择*循环不变式*（或称*归纳断言*），也就是每次进入循环中某个特定点时都为真的命题S。然后通过对以某种方式衡量循环次数的参数进行归纳，证明该命题S。例如，该参数可以是我们到达某`while`循环测试的次数，也可以是`for`循环中循环下标的值，还可以是某个涉及每次循环时都递增1的程序变量的表达式。

✦ 示例 2.12

举个例子，我们考虑一下2.2节中`SelectionSort`的内层循环。以下这几行代码带着与图2-2中相同的编号：

```
(2)        small = i;
(3)        for (j = i+1; j < n; j++)
(4)            if (A[j] < A[small])
(5)                small = j;
```

回想一下，这几行代码的目的是使`small`等于`A[i..n-1]`中值最小的元素的下标。要证实该声明为何为真，考虑一下如图2-10所示的该循环的流程图。该流程图展示了执行该程序必需的5个步骤。

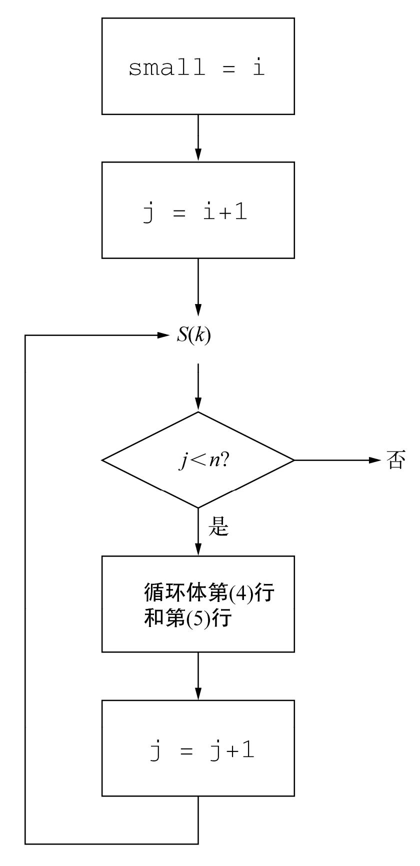


图2-10　`SelectionSort`内层循环的流程图

(1) 首先，需要将`small`初始化为i，如同在第(2)行中所做的那样。

(2) 在第(3)行的`for`循环开始的时候，要将`j`初始化为$i+1$。

(3) 接着，需要测试是否有$j<n$。

(4) 如果是，就执行有第(4)行和第(5)行组成的循环体。

(5) 在循环体结束的位置，需要递增`j`，并返回测试的位置。

在图2-10中看到，在测试之前有一点被标记为循环不变式命题$S(k)$，我们很快就会发现这是个什么样的命题。第一次到达该测试时，`j`的值为$i+1$，而`small`的值为i。第二次到达该测试时，`j`的值是$i+2$，因为`j`已经递增了一次。因为循环体（第4行和第5两行）会在$A[i+1]$比$A[i]$小的条件下将`small`置为$i+1$，所以我们看到`small`总是$A[i]$和$A[i+1]$中较小的那个的下标。[①]

① 为防止出现持平的情况，`small`应该是i。不过一般情况下我们会假设不会出现持平的情况，并将“最小元素第一次出现”说成“最小的元素”。

类似地，当第三次到达测试时，j的值是$i+3$，而small则是A[i..i+2]中最小那个的下标。因此我们将试着证明看似一般规则的如下命题。

命题$S(k)$。在k为循环下标j的值的情况下，如果到达第(3)行的for声明中对$j<n$的测试，那么small的值就是A[i..k-1]中最小元素的下标。

请注意，我们在这里使用字母k来表示变量j在循环进行时可能具有的值。这不像用j来表示j的值那样烦琐，因为有时候需要保持k不变，而同时j的值又在变化。还要注意，$S(k)$的表述中有"如果到达……"，这是因为某些k的值比循环下标j的值更小而使循环中断，所以可能根本没法到达循环测试。如果k是这些值之一，那么$S(k)$一定为真，因为任何"若A则B"形式的命题在A为假时都为真。

依据。依据情况是$k=i+1$，其中i为第(3)行中变量i的值。[①]在循环开始时，有$j=i+1$。也就是说，我们刚执行完第(2)行，把i赋值给small，并且将j初始化为$i+1$，以开始该循环。$S(i+1)$表示，small是A[i..i]中最小元素的下标，也就是说small的值一定是i。从技术上讲，我们还必须证明，除了第一次到达测试时之外，j的值从不可能是$i+1$。从直观上说，其原因就是每次进行循环时，j都会递增，所以它再也不会有$i+1$这么小了。（为了精益求精，我们应该在除了第一次通过测试外都有$j>i+1$的假设下进行归纳证明。）因此，归纳依据$S(i+1)$被证明为真。

归纳。现在假定我们的归纳假设$S(k)$对某些$k\geqslant i+1$成立，并证明$S(k+1)$为真。首先，如果$k\geqslant n$，那么在j的值为k，或更早之前，循环就中断了，所以肯定不会在j的值等于$k+1$时到达该循环测试。在这种情况下，$S(k+1)$一定为真。

因此，我们假设$k<n$，如此一来，实际上已经进行了j等于$k+1$时的测试。$S(k)$说的是small表示A[i..k-1]中最小元素的下标，而$S(k+1)$则是说small表示A[i..k]中最小元素的下标。如果考虑当j的值为k时循环体（第4行和第5行）中会发生的事情，就会出现如下两种情况，具体取决于第(4)行的测试是否为真。

(1) 如果$A[k]$不小于A[i..k-1]中的最小值，那么small的值不会改变。不过，在这种情况下，small还要表示A[i..k]中最小元素的下标，因为$A[k]$不是最小的。因此，在这种情况下$S(k+1)$表述的结论为真。

(2) 如果$A[k]$小于$A[i]$到$A[k-1]$这些值的最小值，那么就要将small置为k。$S(k+1)$表述的结论还是成立，因为k是A[i..k]中最小元素的下标。

因此，不管哪种情况，small都是A[i..k]中最小元素的下标。我们通过递增变量j来进行for循环。因此，在循环测试之前，当j的值为$k+1$时，$S(k+1)$表述的结论成立。现在就证明了由$S(k)$可以得到$S(k+1)$。我们已经完成了归纳，并得到$S(k)$对所有$k\geqslant i+1$的值都为真这样的结论。

接下来，应用$S(k)$来声明第(3)行到第(5)行的内层循环。当j的值达到n时，程序会退出循环。因为$S(n)$表示small是A[i..n-1]中最小元素的下标，所以可以得出一个有关内层循环工作方式的重要结论。我们会在下一个示例中看看如何利用这个结论。

✦ 示例 2.13

现在，考虑整个SelectionSort函数，我们在图2-11中重现了其核心部分。表示这段代码的流程图如图2-12所示，其中"循环体"是指图2-11中的第(2)到(8)这几行。归纳断言$T(m)$还

① 就行(3)到行(5)的循环而言，i是不会改变的。因此i+1是可用作根据值的合适常数。

是关于在循环终止的测试开始前什么一定为真的命题。通俗地说，就是当`i`的值为m时，我们选中较小的m个元素，并将它们排序在数组开头的位置。更具体地讲就是，我们要通过对m的归纳证明以下命题$T(m)$为真。

```
(1)         for (i = 0; i < n-1; i++) {
(2)             small = i;
(3)             for (j = i+1; j < n; j++)
(4)                 if (A[j] < A[small])
(5)                     small = j;
(6)             temp = A[small];
(7)             A[small] = A[i];
(8)             A[i] = temp;
            }
```

图2-11　`SelectionSort`函数的主体

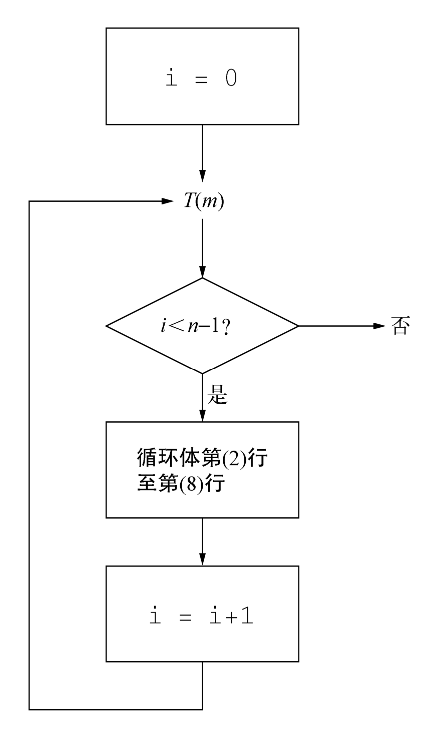


图2-12　整个选择排序函数的流程图

命题$T(m)$。如果到达第(1)行中$i<n-1$的循环测试时变量`i`的值等于m，那么有：

(a) `A[0..m-1]`是有序排列的，也就是说，$A[0]\leqslant A[1]\leqslant\cdots\leqslant A[m-1]$。

(b) `A[m..n-1]`的所有元素不小于`A[0..m-1]`中任一元素。

依据。依据情况是$m=0$。依据为真的原因微不足道。如果考虑命题$T(0)$，那么(a)部分就是说`A[0..-1]`是已排序的。不过在$A[0]$、…、$A[-1]$的范围内没有元素，所以(a)一定为真。类似地，$T(0)$的(b)部分是说，`A[0..n-1]`的所有元素都至少与`A[0..-1]`中任一元素一样大。由于后者描述的范围内没有元素，所以(b)部分也为真。

归纳。在归纳步骤中，假设$T(m)$对所有的$m\geqslant 0$都为真，并要证明$T(m+1)$成立。就像在示例2.12中那样，我们又要试着证明形如“若A则B”的命题，而只要A为假，那么这样的命题肯

定为真。因此，如果“到达for循环测试时i等于$m+1$”这一假设为假，那么$T(m+1)$就为真。因而可以假设我们确实在i的值为$m+1$时到达了该测试，也就是说，可以假设$m<n-1$。

当i的值为m时，循环体会找出A[m..n-1]中的最小元素（如示例2.12中命题$S(m)$所证明的）。在第(6)行至第(8)行中，该元素会与$A[m]$互换。归纳假设$T(m)$的(b)部分告诉我们，被选中的这个元素不小于A[0..m-1]中任一元素。此外，那些元素还是已排序的，所以现在A[i..m]中所有元素也是已排序的。这也就证明了命题$T(m+1)$的(a)部分。

要证明$T(m+1)$的(b)部分，我们看到所选择的$A[m]$不大于A[m+1..n-1]中的任一元素。$T(m)$的(a)部分告诉我们，A[0..m-1]已经不大于A[m+1..n-1]中任一元素了。因此，在执行函数的第(2)行到第(8)行并递增i后，可知A[m+1..n-1]中所有元素都不小于A[0..m]中任一元素。由于现在i的值为$m+1$，我们证明了命题$T(m+1)$的真实性，所以就证明了该归纳步骤。

现在，令$m=n-1$。我们知道，当i的值为$n-1$时，会退出外层循环，所以$T(n-1)$将会在完成这次循环后成立。$T(n-1)$的(a)部分表示，A[0..n-2]中所有元素都是已排序的，而其(b)部分则是说$A[n-1]$不小于其他任何元素。因此，在该程序终止后，A中的元素是以非递减顺序排列的，也就是说，它们是已排序的。

2.5.2　while循环的循环不变式

在讲到形如

```
while (<condition>)
    <body>
```

的while循环时，通常都可以为循环条件测试前的那一点找出合适的循环不变式。一般来说，我们会试着通过对循环次数的归纳来证明循环不变式成立。然后，当条件为假时，可以利用循环不变式以及条件为假的事实，得出一些关于while循环终止后什么为真的有用信息。

不过，与for循环不同的是，可能不存在为while循环计数的变量。更糟的是，尽管for循环可以保证最多只会迭代到循环的限制（例如，SelectionSort程序的内层循环最多循环$n-1$次），我们却没理由相信while循环的条件可能会变为假。因此，证明while循环正确性的部分工作就是要证明while循环最终会终止。一般要通过涉及程序中变量的某个表达式E，按照如下方式一起来证明循环的终止。

(1) 每进行一次循环，E的值至少会减少1。

(2) 如果E的值小到某个指定的常数（比如0），循环条件就为假。

✦　示例 2.14

阶乘函数，写作$n!$，表示的是整数$1\times2\times\cdots\times n$的积。例如，$1!=1$，$2!=1\times2=2$，$5!=1\times2\times3\times4\times5=120$。图2-13所示的简单程序片段就是用来计算$n\geqslant1$时的$n!$的。

```
(1)        scanf("%d", &n);
(2)        i = 2;
(3)        fact = 1;
(4)        while (i <= n) {
(5)            fact = fact*i;
(6)            i++;
           }
(7)        printf("%d\n", fact);
```

图2-13　阶乘程序片段

首先，要证明图2-13中第(4)行至第(6)行的while循环一定会终止。这里我们选择的表达式E是$n-i$。请注意，每进行一次while循环，i的值在第(6)行会增加1，而n的值则保持不变。因此，每进行一次循环，E的值就会减少1。此外，当E的值为-1或更小时，有$n-i \leqslant -1$，或者说是$i \geqslant n+1$。因此，当E变为负值时，循环条件$i \leqslant n$就不再成立，循环就将终止。一开始并不知道E有多大，因为不知道要读入的n值为多少。不过，不管该值为多少，E最终都能小到-1，而循环将会终止。

现在必须证明图2-13中的程序能够完成它应该做的工作。合适的循环不变式命题如下，我们要通过对变量i的值的归纳来证明该命题。

命题 $S(j)$。如果在到达循环测试$i \leqslant n$时变量i的值为j，那么变量fact的值就是$(j-1)!$。

依据。归纳依据是$S(2)$。只有当从外部进入该循环时，在到达该测试时i的值才为2。在循环开始前，图2-13中的第(2)行和第(3)行会将fact的值置为1，并将i的值置为2。由于$1=(2-1)!$，所以归纳依据得到证明。

归纳。假设$S(j)$为真，并证明$S(j+1)$为真。如果$j>n$，那么当i的值为j或更早之时，该while循环就中断了，因此当i的值为$j+1$时，我们根本无法到达该循环测试。在这种情况下，$S(j+1)$为平凡真（trivially true），因为它具有“如果我们到达……”这种形式。

因此，假设$j \leqslant n$，并考虑一下在i的值为j时，执行while循环的循环体会发生什么。通过归纳假设，在第(5)行被执行之前，fact的值为$(j-1)!$，而i的值为j。因此，在第(5)行执行完之后，fact的值为$j \times (j-1)!$，也就是$j!$。

在第(6)行，i增加了1，其值就达到了$j+1$。因此，当i带着值$j+1$到达该循环测试时，fact的值是$j!$。命题$S(j+1)$就是说，当i等于$j+1$时，fact等于$\big((j+1)-1\big)!$，也就是$j!$。因此，我们证明了命题$S(j+1)$，并完成了归纳步骤。

之前已经证明了while循环将终止。由此可见，它将在i第一次具有大于n的值时终止。因为i是整数，而且每进行一次循环就会增加1，所以i在循环终止时的值一定是$n+1$。因此，当到达第(7)行时，命题$S(n+1)$一定成立。不过该命题表示fact的值为$n!$。因此，程序会打印出$n!$，正如我们想要证明的。

作为一个实际问题，应该指出，图2-13中的阶乘程序在任何计算机上都只能打印出少量几个n的阶乘值$n!$作为答案。因为阶乘函数增长得特别快，答案的大小很快就超过了现实中任何一台计算机上整数的最大大小。

2.5.3　习题

(1) 以下程序片段会让sum的值等于从1到n的整数之和，为其找出合适的循环不变式。

```
scanf("%d",&n);
sum = 0;
for (i = 1; i <= n; i++)
    sum = sum + i;
```

通过对i的归纳证明找出的循环不变式成立，并利用它证明程序可按照预期工作。

(2) 以下程序片段可计算数组A[0..n-1]中各整数之和：

```
sum = 0;
for (i = 0; i < n; i++)
    sum = sum + A[i];
```

为其找出合适的循环不变式，利用该循环不变式证明程序可按照预期工作。

(3) * 考虑如下片段：

```
scanf("%d", &n);
x = 2;
for (i = 1; i <= n; i++)
    x = x * x;
```

对应 $i \leqslant n$ 的测试之前那点的合适循环不变式会满足如下条件：如果我们到达该点时变量i的值为k，那么有 $x = 2^{2^{k-1}}$ 。通过对k的归纳，证明该不变式成立。在循环终止后，x的值是多少？

(4) * 图2-14中的程序片段会持续读入整数，直到读到负整数为止，然后会打印出这些整数的和。为循环测试之前的那点找出合适的循环不变式，利用该不变式证明该程序片段可按照预期工作。

```
sum = 0;
scanf("%d", &x);
while (x >= 0) {
    sum = sum + x;
    scanf("%d", &x);
}
```

图2-14 为一列整数求和，通过负整数来终止循环

(5) 考虑图2-13所示程序中的n，找出自己的计算机能处理的n的最大值。定长整数对证明程序的正确有什么影响？

(6) 通过对图2-10中程序循环的次数进行归纳，证明在第一次循环后 $j > i + 1$ 。

2.6 递归定义

在*递归定义*（或*归纳定义*）中，我们用一类或多类紧密相关的对象或事实本身来对它们进行定义。这种定义一定不能是无意义的，比如“某个部件是某个有某种颜色的部件”，也不能是似是而非的，比如“当且仅当某事物不是glotz时它才是glotz”。归纳定义涉及：

(1) 一条或多条*依据规则*，在这些规则中，要定义一些简单的对象；

(2) 一条或多条*归纳规则*，利用这些规则，通过集合中较小的对象来定义较大的对象。

✦ 示例 2.15

在2.5节中我们通过迭代算法定义了阶乘函数：将$1 \times 2 \times \cdots \times n$ 相乘得到$n!$。其实，还可以按照以下方式递归地定义$n!$的值。

依据。$1! = 1$。

归纳。$n! = n \times (n-1)!$。

例如，依据告诉我们$1! = 1$。这样就可以在归纳步骤中使用该事实，得到 $n = 2$ 时

$$2! = 2 \times 1! = 2 \times 1 = 2$$

对n =3、4和5，有

$$3! = 3 \times 2! = 3 \times 2 = 6$$
$$4! = 4 \times 3! = 4 \times 6 = 24$$
$$5! = 5 \times 4! = 5 \times 24 = 120$$

等等。请注意，虽然术语“阶乘”看起来就是用自身来定义的，但在实践中，可以只通过值较小的n的阶乘，得到值逐步增大的n对应的$n!$的值。因此，我们具备了有意义的“阶乘”定义。

严格地讲，应该证明，$n!$的递归定义可以得出与原来的定义相同的结果，

$$n! = 1 \times 2 \times \cdots \times n$$

要证明这一点，就要证明如下命题。

命题 $S(n)$。按照上述方式递归地定义的$n!$，等于$1 \times 2 \times \cdots \times n$。

通过对n的归纳来完成证明。

依据。$S(1)$显然成立。递归定义的依据告诉我们$1! = 1$，而且$1 \times \cdots \times 1$（即“从1到1”这些整数的积）的积显然也等于1。

归纳。假设$S(n)$成立，也就是说，由递归定义给出的$n!$等于$1 \times 2 \times \cdots \times n$。而递归定义告诉我们

$$(n+1)! = (n+1) \times n!$$

如果应用乘法交换律，就有

$$(n+1)! = n! \times (n+1) \tag{2.11}$$

由归纳假设可知

$$n! = 1 \times 2 \times \cdots \times n$$

因此，可以用$1 \times 2 \times \cdots \times n$替换等式(2.11)中的$n!$，就可以得到

$$(n+1)! = 1 \times 2 \times \cdots \times n \times (n+1)$$

这也就是命题$S(n+1)$。这样就证明了归纳假设，并证明了对$n!$的递归定义与迭代定义是相同的。

图2-15显示了递归定义的一般本质。它在结构上与完全归纳类似，都含有无限的实例序列，每个实例都依赖于之前的任一或所有实例。我们通过应用一个或多个依据规则开始。接下来的一轮归纳，要对已经得到的内容应用一条或多条归纳规则，从而建立新的事实或对象。再接下来的一轮归纳，再次对已经掌握的内容应用归纳规则，获得新的事实或对象，以此类推。

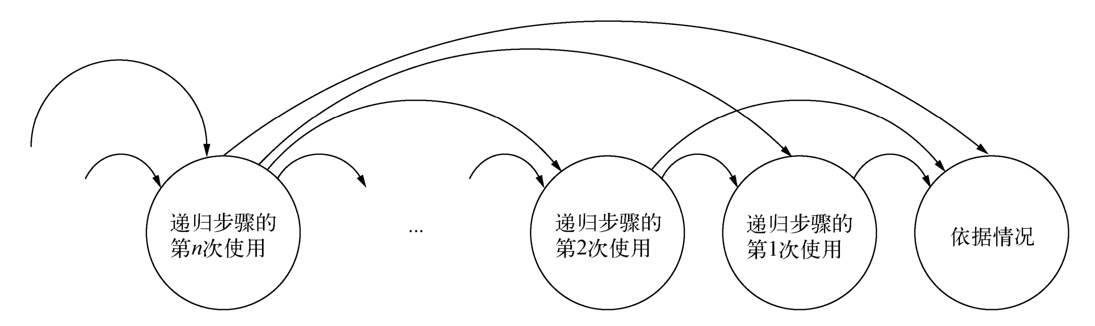


图2-15 在归纳定义中，要一轮一轮地建立对象，在某一轮建立的对象可能会依赖于之前所有轮次建立的对象

在定义阶乘的示例2.15中，我们从依据情况得到了1!的值，应用一次归纳步骤得到2!，应用两次归纳步骤得到3!，等等。这里的归纳具有“普通”归纳的形式，在每一轮的归纳中，都只用到在前一轮归纳中得到的内容。

✦ 示例 2.16

在2.2节中，定义了词典顺序的概念，当时的定义是具有迭代性质的。粗略地讲，通过从左起比较对应符号c_i和d_i，测试字符串$c_1 \cdots c_n$是否先于字符串$d_1 \cdots d_m$，直到找到某个值i令$c_i \neq d_i$，或者到达其中一个字符串的结尾。以下的递归定义定义了字符串对w和x，其中w在词典顺序上

要先于x。从直观上讲，要对两个字符串从开头起相同字符对的数目进行归纳。

依据。该依据涵盖了那些我们能立即分出字符顺序先后的字符串对。依据包含如下两个部分。

(1) 对任何不为ϵ的字符串w，都有$\epsilon<w$。回想一下，ϵ表示空字符串，也就是不含字符的字符串。

(2) 如果$c < d$，其中c和d都是字符，那么对任何字符串w和x，都有$cw < dx$。

归纳。如果字符串w和x具有$w < x$的关系，那么对任意字符c，都有$cw < cx$。

例如，可以使用以上定义表明base < batter。根据依据的规则(2)，有c = s，d = t，w =e，x = tter，因此有se < tter。如果应用递归规则一次，有c = a，w =e，以及x = tter。最后，第二次应用递归规则，有c = b，w =ase，以及x = atter。也就是说，依据和归纳步骤是如下这样的：

```
se      <    tter
ase     <    atter
base    <    batter
```

还可以按照以下方式证明bat < tter。依据的部分(1)告诉我们，ϵ < ter。如果应用递归规则3次，其中c依次等于t、a和b，就可以进行如下推理：

```
ϵ       <    ter
t       <    tter
at      <    atter
bat     <    batter
```

现在应该对两个字符串从左端起相同的字符数进行归纳，证明当且仅当字符串按照刚给出的递归定义排在前面之时，才能按照2.2节中的定义得出它也排在前面的结论。我们还留了两个归纳证明的习题。

在示例2.16中，如图2-15所示的事实组是很大的。依据情况给出了所有$w < x$的事实，不管是w=ϵ，还是w和x以不同字符开头。使用归纳步骤一次，就给出当w和x只有第一个字母相同时，所有$w < x$的情况；第二次使用，就给出了那些当w和x只有前两个字母相同时的所有情况，以此类推。

2.6.1 表达式

各种算术表达式是递归定义的，我们为这种定义的依据指定了原子操作数可以是什么。例如，在C语言中，原子操作数既可以是变量，也可以是常量。然后，归纳过程告诉我们可应用哪些运算符，以及每个运算符可以应用到多少个操作数上。例如，在C语言中，运算符 < 可以应用到两个操作数上，运算符符号−可以应用于一至两个操作数，而由一对圆括号加上括号内必要数量的逗号表示的函数应用运算符，则可以应用于一个或多个操作数，比如$f(a_1, \cdots, a_n)$。

✦ 示例 2.17

通常将如下的表达式称作“算术表达式”。

依据。以下类型的原子操作数是算术表达式：

(1) 变量；

(2) 整数；

(3) 实数。

归纳。如果E_1和E_2是算术表达式，那么以下表达式也是算术表达式：

(1) （E_1+E_2）

(2)（$E_1 - E_2$）

(3)（$E_1 \times E_2$）

(4)（E_1 / E_2）

运算符+、−、×和/都是二元运算符，因为它们都接受两个参数。它们也叫作中缀（插入）运算符（infix operator），因为它们出现在两个参数之间。

此外，我们允许减号在表示减法之外，还可以表示否定（符号改变）。这种可能性反映在了第5条，也是最后一条递归规则中：

(5) 如果E是算术表达式，那么 $(-E)$ 也是。

像规则(5)中的−这样只接受一个操作数的运算符，称为一元运算符。它也称为前缀运算符，因为它出现在参数之前。

图2-16展示了一些算术表达式，并解释了为什么它们都是表达式。请注意，有时候括号是不必要的，可以忽略。比如图2-16中最后的表达式 (vi)，外层括号和表达式 $-(x+10)$ 两侧的括号都是可以忽略的，而我们可以将其写为 $y-(x+10)$。然而，剩下的括号是必不可少的，因为 $y \times -x+10$ 按照约定会被解释为 $(y \times -x)+10$，这就不是与 $y \times -(x+10)$ 等价的表达式了（例如，试试 $y=1$ 和 $x=1$）。[1]

表达式	说明
(i) x	依据规则(1)
(ii) 10	依据规则(2)
(iii) $(x+10)$	对(i)和(*ii*)应用递归规则(1)
(iv) $\big(-(x+10)\big)$	对(iii)应用递归规则(5)
(v) y	依据规则(1)
(vi) $\big(y \times \big(-(x+10)\big)\big)$	对(v)和(iv)应用递归规则(5)

图2-16　一些算术表达式示例

更多运算符术语

出现在其参数之后的一元运算符，比如表达式$n!$中的阶乘运算符!，称为后缀运算符。如果接受多个操作数的运算符重复地出现在其所有参数之前或之后，那么它们也可以是前缀或后缀运算符。在C语言或普通算术中没有这类运算符的例子，不过我们在5.4节中将要讨论一些所有运算符都是前缀或后缀运算符的表示法。

接受3个参数的运算符就是三元运算符。举例来说，在C语言中，表示“若c则x，否则y”的表达式`c?x:y`中，运算符`?:`就是三元运算符。如果运算符接受k个参数，就称其是k元的。

2.6.2　平衡圆括号

可以出现在表达式中的圆括号串称为平衡圆括号。例如，在图2-16的表达式(vi)中出现的$\big((())\big)$模式，以及如下表达式

[1] 如果运算符约定俗成的优先级（一元的减号优先级最高，接着是乘号和除号，再接着是加号和减号），以及“左结合性”的传统约定（即优先级相同的运算符——比如一串加号和减号——从左边开始结合）已经暗示了括号的存在，那么括号就是多余的。不管是C语言还是普通的算术，都遵守这些约定。

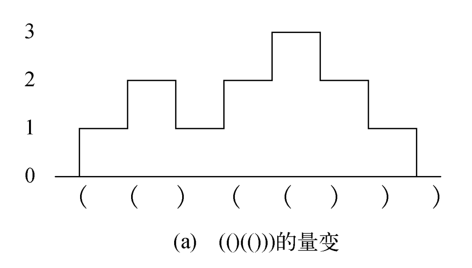
$$\big((a+b)\times((c+d)-e)\big)$$

具有的$\big(()(())\big)$模式。空字符串ϵ也是平衡圆括号串，例如，它的模式是表达式x。一般来说，判定圆括号串平衡的条件是，每个左圆括号都能与其右侧的某个右圆括号配对。因此，“平衡圆括号串”的一般定义由以下两个规则组成：

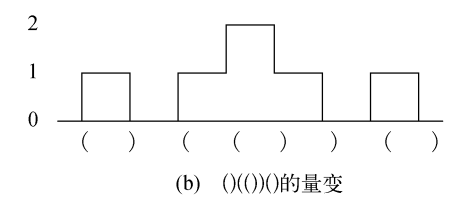
(1) 平衡圆括号串中左圆括号和右圆括号的数量相等；

(2) 在沿着括号串从左向右行进的过程中，该串的量变从不为负值，其中量变（profile）是对行进过程中已达到左括号数目减去已到达右括号数目的累计值。

请注意，统计值必须从0开始，以0结束。例如，图2-17a表示的是$\big(()(())\big)$的量变，而图2-17b表示的是$()(())()$的量变。



(a) (()(()))的量变



(b) ()(())()的量变

图2-17 两个括号串的量变

“平衡圆括号”的概念有着多种递归定义。下面的定义比较巧妙，不过我们将证明，该定义相当于之前提到的涉及统计值的非递归定义。

依据。空字符串是平衡圆括号串。

归纳。如果x和y是平衡圆括号串，那么$(x)y$也是平衡圆括号串。

✦ 示例 2.18

由依据可知，ϵ是平衡圆括号串。如果应用递归规则，其中x和y都等于ϵ，就可以得出()是平衡的。请注意，在将空字符串提交给变量（如x或y）时，该变量就“消失”了。然后可以按以下方法应用递归规则。

(1) $x = ()$且$y = \epsilon$，得出(())是平衡的。

(2) $x = \epsilon$且$y = ()$，得出()()是平衡的。

(3) $x = y =()$，得出(())()是平衡的。

最后，因为已知(())和()()是平衡的，所以可以令递归规则中的x和y为这两者，就证明了((()))()() 是平衡的。

可以证明两种“平衡”定义指定的是同一组括号串。为了让表述更清楚，我们将根据递归

定义定义的平衡括号串直接称为平衡括号串，而将根据非递归定义定义的平衡括号串称为量变平衡括号串。量变平衡括号串就是那些量变最终为0而且从不为负值的括号串。需要证明以下两点。

(1) 每个平衡括号串都是量变平衡的。

(2) 每个量变平衡括号串都是平衡的。

这就是下面两个示例中归纳证明的目标。

✦ 示例 2.19

首先，我们来证明(1)部分，也就是每个平衡括号串都是量变平衡的。这段证明复制了定义平衡括号串所使用的完全归纳。也就是说，我们要证明如下命题。

命题 $S(n)$。如果括号串w是通过n次应用递归规则被定义为平衡的，w就是量变平衡的。

依据。依据为 $n=0$。不需要通过应用任何递归规则便可证明其平衡的括号串就是ϵ，它的平衡是由依据规则得出的。由此可见，空字符串的量变最终为0，而且从不为负值，所以ϵ是量变平衡的。

归纳。假设 $S(i)$ 对 $i=0,1,\cdots,n$为真，并考虑 $S(n+1)$ 的实例，也就是说证明 w 为平衡括号串需要 $n+1$ 次使用递归规则。考虑最后那次递归规则的使用，就是拿两个已知为平衡的括号串x和y，组成形为 $(x)y$ 的w。我们使用了 $n+1$ 次递归规则来形成w，而且最后一次利用递归规则既不是用来形成x，也不是用来形成y。因此，形成x和y都不需要利用递归规则n次以上。所以，归纳假设可以应用于x和y，而且可以得出x和y都是量变平衡的。

w的量变如图2-18所示。它首先会上升一级，作为对第一个左圆括号的回应。接着是x的量变，由虚线所示，w的量变在这里会再上升一级。我们使用归纳假设得出x是量变平衡的，因此，它的量变始于第0级且终于第0级，而且从不为负。如图2-18所示，由于w的量变中x的部分已经上升了一级，该部分从第1级开始，在第1级结束，而且从来不低于第1级。

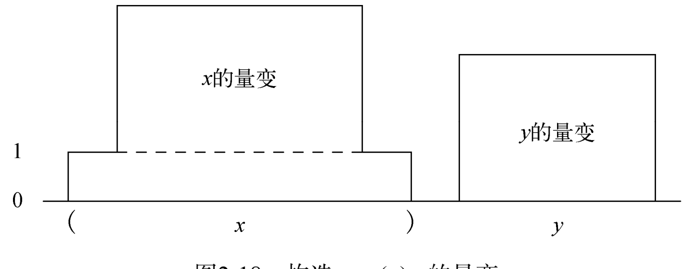


图2-18　构造 $w=(x)y$ 的量变

显式出现在x和y之间的右圆括号将w的量变降为0。接着就到了y的量变。根据归纳假设，y是量变平衡的，因此在w的量变中，y的部分不会低于0，而且它让w的量变最终归于0。

我们现在已经构造了w的量变，并发现它满足量变平衡括号串的条件。也就是说，w的量变从0开始，以0结束，并且从不为负值。这样就证明了，如果括号串是平衡的，那么它就是量变平衡的。

现在介绍“平衡圆括号”两种定义等价性的第二个方向。在示例2.20中，将要证明量变平衡的括号串是平衡的。

对递归定义的证明

请注意，在示例2.19中，通过对递归规则（用来证实某对象在定义的类中）的使用次数进行归纳，证明了与一类递归定义的对象（平衡圆括号串）有关的断言。这是处理递归定义概念的一种常见方法，其实也是递归定义很实用的原因之一。在示例2.15中，通过对n的归纳，证明了递归定义的阶乘值的属性（即$n!$就是1到n这n个整数的积）。而在对$n!$的定义中，使用了$n-1$次递归规则，所以该证明过程也可视作对递归规则应用次数进行归纳。

✦ 示例 2.20

现在来证明(2)部分，通过对圆括号串的长度进行完全归纳，由"量变平衡"得出"平衡"。正式的命题如下。

命题 $S(n)$。如果长度为n的圆括号串w是量变平衡的，那么它也是平衡的。

依据。如果$n=0$，那么该括号串一定是ϵ。由递归定义的依据可知ϵ是平衡的。

归纳。假设长度小于等于n的量变平衡括号串是平衡的。必须证明$S(n+1)$为真，也就是要证明长度为$n+1$的量变平衡括号串也是平衡的。[①]考虑这样一个括号串w：因为w是量变平衡的，它不可能以右圆括号开头，否则它的量变会立刻变为负值。因此，w是以左圆括号开始的。

将w分为两部分。第一部分从w的开头开始，到w的量变第一次变为0截止。第二部分就是w中其余的部分。例如，图2-17a所示的量变第一次变为0是在其末尾处，所以如果$w=(()(()))$，那么第一部分就是整个括号串，而第二部分就是ϵ。在图2-17b中，$w=()(())()$，那么第一部分就是$()$，而第二部分是$(())()$。

第一部分永远不可能以左圆括号结尾，因为如果那样，那么在结尾之前的那个位置，量变就为负值了。因此，第一部分以左圆括号开始，并以右圆括号结尾。这样就可以将w写为$(x)y$的形式，其中(x)是第一部分，而y是第二部分。x和y都要比w短，所以如果可以证明它们是量变平衡的，就可以利用归纳假设推出它们是平衡的。然后可以使用"括号串平衡"定义中的递归规则来证明$w=(x)y$是平衡的。

很容易看出，y是量变平衡的。图2-18还说明了w、x和y的量变之间的关系。也就是说，y的量变是w的量变的尾部，开始和结束的高度都是0。因为w是量变平衡的，所以可以得出结论：y也是量变平衡的。证明x是量变平衡括号串的过程也几近相同。x的量变是w的量变的一部分，它的起止高度都是第1级，而且x的量变也从未低于第1级。可以知道w的量变在x这一段从未到过0，因为我们选取(x)作为w的最短前缀，而在它结尾处w的量变才回到0。这样，w内的x的量变从未到过0，所以x本身的量变从未变为负值。

现在已经证明了x和y都是量变平衡的。因为它们都比w短，所以归纳假设适用于它们，它们都是平衡的。定义"括号串平衡"的递归规则告诉我们，如果x和y都是平衡的，那么$(x)y$也是平衡的。而$w=(x)y$，所以w也是平衡的。我们现在完成了归纳步骤，并证明了命题$S(n)$对所有的$n\geqslant 0$都成立。

① 请注意，所有的量变平衡括号串都刚好是偶长度的，所以，如果$n+1$为奇数，就不作说明了。不过，在证明中不需要n为偶数。

2.6.3　习题

(1) * 证明：示例2.16中给出的词典顺序的定义和2.2节中给出的定义是相同的。*提示*：证明由两部分组成，每个部分都要通过归纳法进行证明。对第一个部分，假设根据示例2.16的定义有 $w<x$ 。通过对i的归纳证明如下命题 $S(i)$ 为真："如果证明 $w<x$ 需要应用i次递归规则，那么根据2.2节中'词典顺序'的定义有，w先于x。"依据情况为 $i=0$ 。该习题的第二部分是要证明，如果根据2.2节中词典顺序的定义有，w先于x，那么根据示例2.16中的定义有， $w<x$ ，这里要对w和x从开头起不间断的相同字母数进行归纳。

(2) 画出以下圆括号串的量变曲线。

(a) (()(())
(b) ()())(()
(c) ((()())()())
(d) (()(()(())))

哪些是量变平衡的？对那些量变平衡的圆括号串，使用2.6节中的递归定义证明它们是平衡的。

(3) * 证明：每个平衡圆括号串（按照2.6节中的递归定义）都是某个算术表达式中的圆括号串（见介绍算术表达式定义的示例2.17）。*提示*：对"平衡圆括号"定义中的递归规则在构建某给定平衡圆括号串的过程中被使用的次数进行归纳，以证明该命题。

(4) 说出以下C语言运算符是前缀、后缀还是中缀运算符，以及它们是一元、二元还是k元（ $k>2$ ）运算符。

(a) `<`
(b) `&`
(c) `%`

(5) 如果熟悉UNIX的文件系统或类似的系统，请对可能的目录/文件结构给出递归定义。

(6) * 某整数集合S可通过以下规则递归地定义。

依据。0在S中。

归纳。如果i在S中，那么 $i+5$ 和 $i+7$ 也在S中。

(a) 不在S中的最大整数是多少？

(b) 设j是(a)小题的答案。证明：不小于 $j+7$ 的整数都在S中。*提示*：要注意到这一题与2.4节习题中第(8)小题的相似性（虽然在这里我们只处理非负整数）。

(7) * 通过对位串长度的归纳，递归地定义偶校验位串集合。*提示*：最好同时定义偶校验位串和奇校验位串这两个概念。

(8) * 可以按照以下规则定义已排序整数表。

依据。由一个整数组成的表是已排序的。

归纳。如果已排序表L的最后一个元素是a，而且 $b \geqslant a$ ，那么L后加上b也是已排序表。

证明：如上所述的对"已排序表"的递归定义与之前对已排序表的非递归定义（即由整数 $a_1 \leqslant a_2 \leqslant \cdots \leqslant a_n$ 组成的表是已排序表）是等价的。

请记住，这里需要证明(a)、(b)两个部分。(a) 如果由递归定义得出表是已排序的，那么根据非递归定义它也是已排序的；(b) 如果表由非递归定义可知是已排序的，那么根据递归定义它也是已排序的。(a)部分可以对递归规则的使用次数进行归纳，而(b)部分则可以对表的长度进行归纳。

(9) ** 如图2-15所示，每当我们给出递归定义，就可以按照生成对象的"轮次"（也就是为得到对象而应用归纳步骤的次数）为已定义的对象分类。在示例2.15和示例2.16中，描述每一轮生成的结果是相当简单的。有时这项工作却更具挑战性。请问该如何刻画以下两种情况中第n轮生成的对象？

(a) 如示例2.17中描述的算术表达式。*提示*：如果熟悉第5章要介绍的树，可以考虑表达式的树表示。

(b) 平衡圆括号串。请注意，示例2.19中所描述的"递归规则应用次数"与找出圆括号串的轮次是不同的。例如，(())()使用了3次递归规则，但是在第二轮被找出的。

2.7　递归函数

递归函数是那些在自己的函数体中被调用的函数。这种调用通常是直接的，例如，函数F在它自己中包含对F的调用。不过，有时候这种调用也会是间接的，比如，某函数F_1直接调用函数F_2，F_2又直接调用F_3，等等，直到该调用链中的函数F_k调用F_1。

人们通常有这样一种看法，就是学习迭代编程，或者说使用非递归的函数调用，要比学习递归编程更容易。诚然，我们不能完全否定这种观点，但我们相信，只要大家有机会对递归编程加以练习，那么它其实也是很简单的。递归程序往往比等效的的迭代程序更简洁且更易于理解。更重要的是，比起迭代程序，某些问题更容易被递归程序击破。[1]

说几句实在话

使用递归存在潜在的缺点，即在某些计算机上对函数的调用会非常费时，因而与解决同样问题的迭代程序相比，递归程序可能会耗时更多。不过，在很多现代化的计算机上，函数调用是非常高效的，所以反对使用递归程序的这一理由已变得不那么重要。

即便是在函数调用机制较慢的计算机上，人们也可以对程序进行剖析，看看程序的各个部分分别花了多少时间。然后就可以重新编写程序中占用大部分运行时间的部分，如有必要就用迭代替代递归。这样一来，除了速度是最关键因素的一小部分代码之外，程序的大半部分都能利用递归。

通常可以通过模仿待实现程序的规范中的递归定义，来设计递归算法。实现递归定义的递归函数将含有一个依据部分与一个归案部分。依据部分一般会检查可由定义的依据解决的简单输入（不需要递归调用）。函数的归纳部分则需要一次或多次对其本身进行递归调用，并实现定义的归纳部分。下面的例子应该能说明这几点。

✦　示例 2.21

图2-19给出了计算某个非负整数n的阶乘值$n!$的递归函数。该函数直接转换了示例2.15中对$n!$的递归定义。也就是说，图2-19的第(1)行依据情况与归纳情况进行了区分。我们假设$n \geqslant 1$，所以第(1)行的测试其实就是在问是否有$n = 1$。如果是，我们就在第(2)行应用依据规则，得到$1! = 1$。如果$n > 1$，就在第(3)行应用归纳规则$n! = n \times (n-1)!$。

```
          int fact(int n)
          {
(1)           if (n <= 1)
(2)               return 1; /*依据*/
              else
(3)               return n*fact(n-1); /*归纳*/
          }
```

图2-19　计算$n \geqslant 1$时$n!$的递归函数

---

① 这样的问题往往涉及某种查找。例如，在第5章中我们会看到一些查找树的递归算法，这些算法没有方便的迭代模拟（虽然也存在使用栈的等价迭代算法）。

例如，如果我们调用fact(4)，结果就会调用fact(3)，然后调用fact(2)，再调用fact(1)。至此，fact(1)会应用依据规则，因为这里有$n \leqslant 1$，并为fact(2)返回值1。这次对fact的调用在第(3)行完成，返回2给fact(3)。接着，fact(3)返回6给fact(4)，而fact(4)最后会在第(3)行返回24作为答案。图2-20表示了这些调用和返回的模式。

```
调用 ↓                                                   ↑返回24
    fact(4)                                               fact(4)
     调用 ↓                                        ↑返回6
         fact(3)                                   fact(3)
           调用 ↓                            ↑返回2
               fact(2)                       fact(2)
                 调用 ↓            ↑返回1
                     fact(1)
```

图2-20　调用fact(4)所带来的调用和返回

防御性程序设计

图2-19中的程序说明了很重要的一点，即编写递归程序时要注意不让它们陷入无限的调用。我们可能暗自假设不会以小于1的参数来调用fact。当然，最好的做法是在调用fact之前测试是否有$n \geqslant 1$，如果n不满足这个条件就打印错误消息并返回某个特定的值（比如0）。不过，即便我们坚信不会以小于1的n来调用fact，也还是要明智一些，在依据情况中包含所有的“错误情况”。这样一来，以错误的输入调用函数fact会直接返回值1，虽然这是不对的，但不至于造成程序出错（其实，对n=0来说，结果为1也是对的，因为0的阶乘等于1）。

然而，假如忽略掉错误情况，并将图2-19的第(1)行写成

```
if(n == 1)
```

那么如果调用了fact(0)，它就会被看作递归情况的实例，并会接着调用fact(-1)、fact(-2)，等等，直到计算机用尽记录递归调用的空间才会出错终止。

我们可以像绘制归纳证明和归纳定义的图那样绘出递归图。在图2-21中，假设存在递归函数的参数“大小”这样一个概念。例如，对示例2.21中的fact函数而言，参数n的值就具有合适的大小。我们将在2.9节中介绍更多与这种大小有关的内容。不过，在这里要注意的是，递归调用只会调用大小更小的参数。还有，在到达某特定大小时（比如在图2-21中就是大小为0），就必须达到依据情况，也就是必须终止递归。

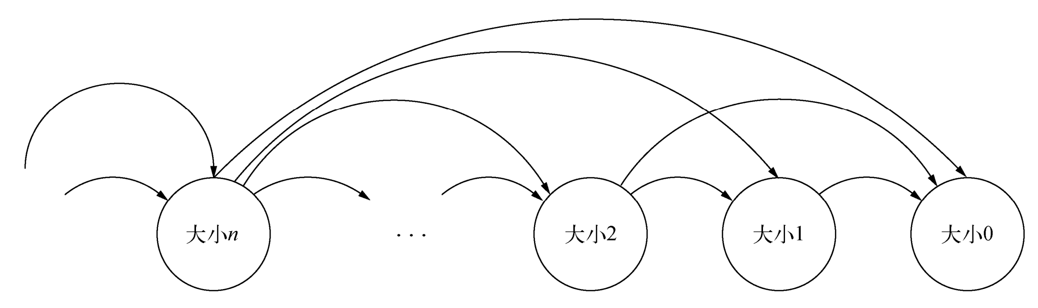


图2-21　递归函数只调用具有更小的参数的本身

在fact函数的例子中，调用过程不像图2-21所示那样具有一般性。调用fact(n)会导致对fact(n-1)的直接调用，但fact(n)不会直接调用其他具有更小参数的fact。

✦ 示例 2.22

如果将底层算法表示为如下形式，就可以将图2-2中的SelectionSort函数变成递归函数recSS。此处假设要排序的数据是在数组A[0..n-1]中。

(1) 从数组A的尾部，也就是从A[i..n-1]中，选出最小的元素。

(2) 将步骤(1)中选出的元素与A[i]互换。

(3) 将剩下的数组A[i+1..n-1]进行排序。

我们可用如下递归算法表示选择排序。

依据。如果$i = n-1$，那么数组中只有一个元素需要排序。因为任意一个元素都是已排序的，所以我们什么都不用做。

归纳。如果$i<n-1$，那么要找出A[i..n-1]中最小的元素，将其与A[i]互换，并递归地将A[i+1..n-1]进行排序。

整个算法是从$i=0$开始执行以上递归的。

如果将i视作上述归纳中的归纳参数，那么它就是*逆向归纳*（backward induction）的例子。我们从参数最大的依据开始，通过归纳规则，用较大参数的实例去解决较小参数的实例，这是种特别好的归纳风格，虽然我们之前并未提及它的可能性。不过，还可以将上述归纳视为普通的，或是说“正向”归纳，只要将数组尾部待排序元素的数目$k=n-i$作为归纳参数即可。

在图2-22中，我们看到了recSS(A,i,n)程序。其第二个参数i是数组A未排序部分第一个元素的下标，第三个参数n是数组A中待排序元素的总数。不难看出，n是小于等于数组A的最大大小的。因此，调用recSS(A,0,n)会为整个数组A[0..n-1]排序。

```
     void recSS(int A[], int i, int n)
     {
         int j, small, temp;
(1)      if (i < n-1) {/* 依据是 i = n-1，在这种情况下，
                 该函数会返回而不改变 */
                 /* 归纳如下 */
(2)              small = i;
(3)              for (j = i+1; j < n; j++)
(4)                  if (A[j] < A[small])
(5)                      small = j;
(6)              temp = A[small];
(7)              A[small] = A[i];
(8)              A[i] = temp;
(9)              recSS(A, i+1, n);
             }
     }
```

图2-22 递归的选择排序

就图2-21而言，$s=n-i$对recSS函数的参数来说是合适的“大小”概念。依据情况是$s=1$，也就是为一个元素排序，不需要发生递归调用。归纳步骤就讲述了如何通过选出最小元素并排序剩下的$s-1$个元素来为s个元素排序。

在第(1)行，我们会测试依据情况，就是只有一个元素需要排序的情况（这里我们再次进行了防御性编程，这样一来，就算在调用时有$i \geqslant n$，也不会造成无限的调用）。在该依据情况中，

我们无事可做，所以直接返回。

函数其余部分是归纳情况。第(2)至(8)行直接照搬了递归选择排序程序中的内容。就像那个程序那样，这几行会将数组`A[i..n-1]`最小元素的下标赋值给`small`，并将该元素与`A[i]`互换。最后，第(9)行是递归调用，会排序数组其余部分。

习题

(1) 我们可以按下以下方式递归地定义 n^2 。

依据。对 $n=1$ ，有 $1^2=1$

归纳。如果 $n^2=m$ ，那么 $(n+1)^2=m+2n+1$ 。

(a) 编写C语言递归函数实现该递归。

(b) 通过对n的归纳证明该定义能正确地计算 n^2 。

(2) 假设有数组`A[0..4]`，其中有按所述顺序排列的元素10、13、4、7、11。在每次调用图2-22所示的递归函数`recSS`之前，数组`A`的内容是怎样的？

(3) 假设如1.3节所述，使用1.6节给出的`DefCell(int,CELL,LIST)`宏来定义整数链表的节点。回想一下，该宏会扩展为如下类型定义：

```
typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
};
```

编写递归函数`find`，接受`LIST`类型的参数，并在某个链表节点含有整数1698作为其元素时返回`TRUE`，如果没有则返回`FALSE`。

(4) 编写递归函数`add`，像习题(3)那样接受`LIST`类型的参数，并返回表中各元素之和。

(5) 使用习题(3)中提到的节点，编写接受整数链表作为参数的递归选择排序函数。

(6) 我们在习题(8)中提出，可以将选择排序一般化，以使用任意的`key`和`lt`函数来比较元素。重新编写递归的选择排序算法以融入这种一般性。

(7) * 给出递归算法，接受整数i，并生成i的二进制表示形式（由0和1组成的序列），其中低位排在前。

(8) * 两个整数i和j的最大公约数（greatest common divisor，GCD）是指能整除i和j的最大整数。例如，$gcd(24,30)=6$ ，而 $gcd(24,35)=1$ 。编写递归函数，接受两个整数i和j，其中 $i>j$ ，并返回 $gcd(i,j)$ 。提示：大家可以使用如下所述的gcd的递归定义，它假设 $i>j$ 。

依据。如果j能整除i，则j是i和j的最大公约数。

归纳。如果j不能整除i，设k是i除以j得到的余数。那么 $gcd(i,j)$ 就和 $gcd(j,k)$ 是相同的。

(9) ** 证明：习题(8)中给出的最大公约数递归定义和它的非递归定义（整除i和j的最大整数）能得出相同结果。

(10) 通常，递归定义可以相当直接地转化为算法。例如，考虑一下示例2.16中给出的字符串“小于”关系的递归定义。编写递归函数，测试两个给定字符串中的第一个字符串是否“小于”另一个字符串。假设字符串是用字符链表表示的。

(11) * 根据2.6节的习题(8)中给出的已排序表的递归定义，创建一种递归排序算法。该算法与示例2.22中的递归选择排序相比如何？

分 治 法

有一种攻克问题的方式，是将问题分解成多个子问题，然后解决这些子问题，并将它们的解决方案结合成整个问题的解决方案。术语分治法就是用来描述这种问题解决技术的。如果这些子问题和原问题相似，那么我们也许能使用相同的函数递归地解决这些子问题。

要让这种技术起作用，有两点要求。首先是子问题必须比原问题简单。其次是在有限次细分之后，必须得到能立即解决的子问题。如果达不到这些条件，递归算法就会一直细分问题，而找不出解决方案。

我们注意到图2-22中的递归函数recSS就满足这两个条件。每当调用该函数，就是对少一个元素的子数列调用该函数，而在对只含一个元素的子数列调用它时，它就会返回而不再继续调用自己。类似地，图2-19中的阶乘程序会在每次调用时调用较小整数，而递归过程会在调用参数到达1时停止。2.8节讨论了分治法更为强大的应用——“归并排序”。在这种排序中，待排序数组的大小减小得非常迅速，因为归并排序在每次递归调用时会将数组大小砍掉一半，而不是减去1。

2.8　归并排序：递归的排序算法

现在要考虑一种名为*归并排序*的与选择排序有着天壤之别的排序算法。递归的方式能最好地描述归并排序，而归并排序展示了分治法的强大，在这种排序方法中，我们通过将问题“分为”大小减半的两个相似问题来为表 $(a_1, a_2, \cdots, a_n)$ 排序。从原则上讲，可以首先将原表分为两个元素任选的大小相等的表，不过在我们开发的程序中，将会将其分为一个含有奇数编号元素的表 $(a_1, a_3, a_5, \cdots)$，以及一个含有偶数编号元素的表 $(a_2, a_4, a_6, \cdots)$。[①]接着单独为大小减半的两个表排序。要完成原表中n个元素的排序，要使用示例2.23中描述的算法来合并两个大小减半的表。

在第3章中，我们将看到，随着待排序表长度n的增加，归并排序所需时间的增长速度要远慢于选择排序所需时间的增长速度。因此，即便递归调用会额外耗费些时间，当n很大时，还是应该优先使用归并排序而不是选择排序。在第3章中我们将分析这两种排序算法的相对性能。

2.8.1　合并

“合并”是指用两个已排序表生成一个只包含这两个表中所有元素的已排序表。例如，假设有表（1,2,7,7,9）和（2,4,7,8），合并后的表就是(1,2,2,4,7,7,7,8,9)。请注意，对未排序的表谈“合并”是没有意义的。

有一种合并两个表的简单方式，就是从表开头开始分析它们。在每一步中，我们找出两个表当前开头位置的两个元素中较小的那个，选择该元素作为合并后的表的下一个元素，并将该元素从它原来所在的表中删除，使该表具有一个新的“首位”元素。虽然我们在两个表开头的元素相同时会选取第一个表开头的元素，但是持平关系的打破是具有任意性的。

✦　示例 2.23

考虑合并以下两个表。

$L_1 = (1, 2, 7, 7, 9)$ 和 $L_2 = (2, 4, 7, 8)$

两个表的第一个元素分别为1和2 。因为1比较小，所以将其选作合并后的表M的第一个元素，并将1从 L_1 中删除，因此新的 L_1 就是 $(2, 7, 7, 9)$。现在，L_1 和 L_2 的第一个元素都是2。可以任选其一。假设采取持平情况下总是从 L_1 中选取元素的策略，那么合并后的表M就变为 $(1, 2)$，表 L_1 变为 $(7, 7, 9)$，而 L_2 仍为 $(2, 4, 7, 8)$。图2-23所示的表格展示了直到 L_1 和 L_2 双双耗尽的整个合并步骤。

① 请记住，“奇数编号”和“偶数编号”指的是元素在表中的位置，而非这些元素的值。

L_1	L_2	M
1，2，7，7，9	2，4，7，8	空
2，7，7，9	2，4，7，8	1
7，7，9	2，4，7，8	1，2
7，7，9	4，7，8	1，2，2
7，7，9	7，8	1，2，2，4
7，9	7，8	1，2，2，4，7
9	7，8	1，2，2，4，7，7
9	8	1，2，2，4，7，7，7
9	空	1，2，2，4，7，7，7，8
空	空	1，2，2，4，7，7，7，8，9

图2-23　合并的例子

我们将会发现，如果把表表示为1.3节所介绍的链表，设计归并算法的工作会更简单。链表将会在第6章中得到更为详细的介绍。接着，要假设表的元素都为整数。因此，每个元素都能表示为一个“单元”，或者说是struct CELL类型的结构体，而表则表示为指向CELL的LIST类型的指针。这些定义都是由我们在1.6节中讨论过的DefCell(int, CELL, LIST)宏来定义的。这种对DefCell宏的使用会扩展为：

```
typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
};
```

每个单元的element字段都含有一个整数，而next字段则含有指向表中下一单元的指针。如果当前的元素是表中最后一个元素，next字段就含有表示空指针的值NULL。然后整列整数就会用指向表第一个单元的指针（即一个LIST类型的变量）来表示。而空表会用值为NULL的变量（而不是指向第一个元素的指针）来表示。

图2-24是归并算法的C语言实现。merge函数接受两个表作为参数，并返回合并后的表。也就是说，形式参数list1和list2是指向两个给定表的指针，而返回值是指向合并后的表的指针。递归算法可描述为如下形式。

```
     LIST merge(LIST list1, LIST list2)
     {
(1)      if (list1 == NULL) return list2;
(2)      else if (list2 == NULL) return list1;
(3)      else if (list1->element <= list2->element) {
             /* 在这里，两个表都不为空，
                而且第一个表的首个元素更小。
                得到的结果就是第一个表的第一个元素，
                后面跟上其余元素的合并。*/
(4)          list1->next = merge(list1->next, list2);
(5)          return list1;
         }
         else { /* list2 的首个元素更小 */
(6)          list2->next = merge(list1, list2->next);
(7)          return list2;
         }
     }
```

图2-24　递归的合并

依据。如果任一表为空，那么另一个表就是所需的结果。这条规则是通过图2-24中的第(1)行和第(2)行实现的。请注意，如果两个表都为空，就将返回`list2`。不过这是正确的，因为这里`list2`的值是`NULL`，而两个空表的结合还是空表。

归纳。如果两个表都不为空，那么每个表都有第一个元素。我们可以将两个表的第一个元素分别称为`list1->element`和`list2->element`，即分别由`list1`和`list2`指向的单元的`element`字段。图2-25展示了这种数据结构。返回的表从含有最小元素的单元开始。该返回表其余的部分由两个表中除这个最小元素之外的所有元素组成。

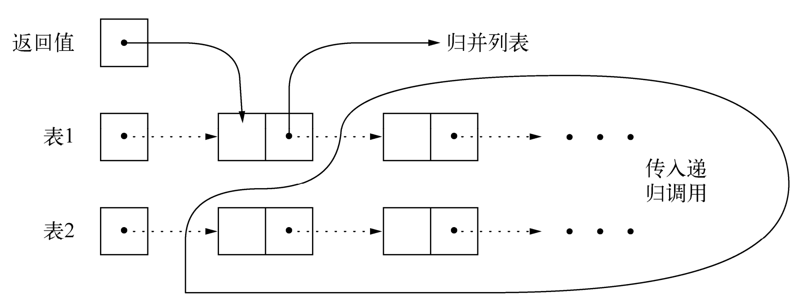


图2-25 归并算法的归纳步骤

例如，第(4)行和第(5)行处理的是最小元素为表1中第一个元素的情况。第(4)行是对`merge`函数的递归调用。该调用的第一个参数是`list1->next`，也就是指向表1中第二个元素的指针（如果表1只有一个元素则为`NULL`）。因此，传入该递归调用的是由表1中除第一个元素之外的所有元素组成的表。第二个参数是整个表2。因此，第(4)行中对`merge`函数的递归调用会返回一个指针，指向合并后的表中其余所有元素，并将该指向合并后的表的指针存储在表1第一个单元的`next`字段中。在第(5)行，我们会返回指向上述单元的指针，该单元现在已经是合并后的表所有元素中的第一个单元。

图2-25展示了这种变化。虚线表示的箭头会在`merge`被调用时出现。特别要说的是，`merge`的返回值是指向最小元素所在单元的指针，而且该元素的`next`字段是指向第(4)行对`merge`的递归调用所返回的表。

最后，第(6)行和第(7)行会处理最小元素在表2中的情况。该算法的行为与第(4)行和第(5)行中的行为是一样的，只不过两个表的角色互换了而已。

✦ 示例 2.24

假设我们对示例2.23中的表 (1,2,7,7,9) 和 (2,4,7,8) 调用`merge`。图2-26展示了进行合并所产生的调用序列，是按照第一列中由上向下的顺序进行调用的。在图中我们省去了分隔表元素的逗号，不过在分隔进行合并的参数时要用到逗号。

调　用	返　回
merge(12779,2478)	122477789
merge(2779,2478)	22477789
merge(779,2478)	2477789
merge(779,478)	477789
merge(779,78)	77789
merge(79,78)	7789
merge(9,78)	789
merge(9,8)	89
merge(9,NULL)	9

图2-26 对`merge`函数的递归调用

例如，因为表1的第一个元素比表2的第一个元素小，所以会执行图2-24的第(4)行，而且我们会归并除表1第一个元素之外的所有元素。也就是说，第一个参数是表1其余的部分，即(2,7,7,9)，而第二个参数就是整个表2，即(2,4,7,8)。现在两个表开头的元素是相同的。因为图2-24中第(3)行的测试偏向表1，所以我们从表中移出2，而对merge函数的下一次调用中，第一个参数就是(7,7,9)，第二个参数还是(2,4,7,8)。

返回的表在第二行中表示，是按从下向上的顺序看的。请注意，与图2-23中合并的迭代描述不同的是，递归算法会从尾部起组成合并后的表，而迭代算法则是从头开始组成合并后的表。

2.8.2　分割表

归并排序的另一项重要任务是将一个表均分为两个表，或者，如果原表的长度为奇数，就分为长度只相差1的两个表。要完成这一工作，一种方式是数出表中元素的数目，然后除以2，并在表的中点将其拆分。我们将给出一个简单的递归函数split，将这些元素“处理”进两个表，其中一个表由第1个、第3个、第5个等元素组成，而另一个表则由偶数位置的元素组成。更确切地说，split函数会将偶数编号的元素从作为参数给出的表中删除，并返回一个由这些偶数编号元素组成的新表。

split函数的C语言代码如图2-27所示，它的参数是LIST类型的表，这样定义是和merge函数有关的。请注意，局部变量pSecondCell被定义为LIST类型。这里是将pSecondCell用作指向表第二个单元（而不是指向表本身）的指针，不过其实LIST类型当然是指向单元的指针。

```
     LIST split(LIST list)
     {
         LIST pSecondCell;

(1)      if (list == NULL) return NULL;
(2)      else if (list->next == NULL) return NULL;
         else { /* there are at least two cells */
(3)          pSecondCell = list->next;
(4)          list->next = pSecondCell->next;
(5)          pSecondCell->next = split(pSecondCell->next);
(6)          return pSecondCell;
         }
     }
```

图2-27　将表均分为两部分

split是个具有副作用的函数。它会从作为参数给出的表中删除偶数位置的单元，而且它会将这些单元组合成一个作为该函数返回值的新表。

我们能以如下形式，用归纳的方式描述该分割算法。它对表的长度进行了归纳，这段归纳具有多个依据情况。

依据。如果表的长度为0或1，那么我们什么都不用做。这就是说，空表会被“分割成”两个空表，而只有一个元素的表，在分割时会将唯一的元素留在给定的表中，并返回一个空的偶数编号元素表（因为原表没有偶数编号的元素，所以这个表中没有元素）。该依据是由图2-27所示程序的第(1)行和第(2)行处理的。第(1)行处理的是list为空的情况，而第(2)行处理的则是list中只含一个元素的情况。请注意，我们在第(2)行中会避免去检查list->next，除非之前在第(1)行中已经确定list不为NULL。

归纳。归纳步骤适用于list中至少存在两个元素的情况。第(3)行中局部变量pSecondCell

中存放了指向表第二个单元的指针；第(4)行则是使第一个单元的next字段跳过第二个单元，直接指向第三个单元，或者，如果表中只有两个单元，就变为NULL；在第(5)行，我们对除前两个元素之外的整个表递归地调用split函数；而split函数会在第(6)行返回一个指向第二个单元的指针，该指针让我们可以访问由原表中所有偶数编号的元素组成的链表。

split带来的变化如图2-28所示。原始指针用虚线表示，而新指针用实线表示。我们还指出了创建每个新指针的代码行编号。

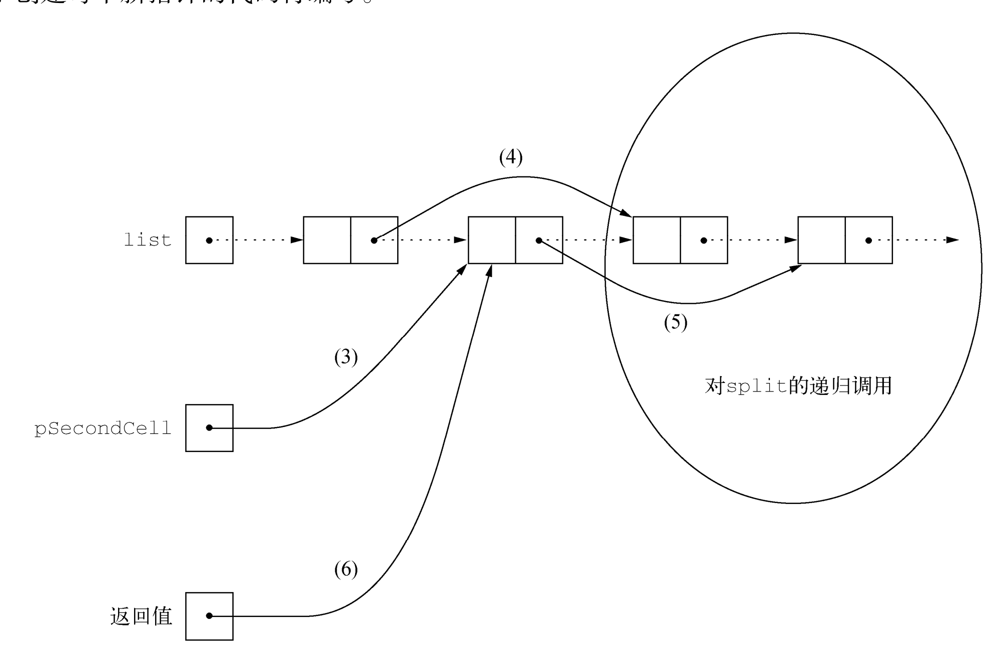


图2-28 split函数的动作

2.8.3 排序算法

递归的排序算法如图2-29所示，该算法可以通过以下依据与归纳步骤来描述。

```
     LIST MergeSort(LIST list)
     {
         LIST SecondList;

(1)      if (list == NULL) return NULL;
(2)      else if (list->next == NULL) return list;
         else {
             /* 表中至少有两个元素 */
(3)          SecondList = split(list);
             /* 请注意，这样做的副作用是有一半元素会从表中删除 */

(4)          return merge(MergeSort(list), MergeSort(SecondList));
         }
     }
```

图2-29 归并排序算法

依据。如果待排序的表为空或长度为1，那么只要返回该表即可，因为它是已排序的。该依据是由图2-29中的第(1)行和第(2)行处理的。

归纳。如果待排序表的长度至少为2，那么在第(3)行使用`split`函数，从`list`中删除偶数编号的元素，并使用这些被删除的元素组成另一个表，该表是由局部变量`SecondList`指向的。第(4)行会递归地为大小减半的表排序，并返回这两个表的归并结果。

✦ 示例 2.25

让我们用归并排序为一列一位数数字742897721排序。为求简洁，我们再次省略了数字之间的逗号。首先，通过`MergeSort`函数第(3)行中对`split`的调用，表会被分为两个部分。生成的两个表中有一个是由奇数位置的元素组成，另一个则由偶数位置的元素组成。也就是说，这里有`list`＝72971，而`SecondList`＝ 4872。在第(4)行，这两个表会被排序，结果就成了表12779和2478，然后就会合并成已排序表122477789。

不过，这两个大小减半的表的排序工作并不是凭空进行的，而是通过对该递归算法的合理应用做到的。一开始，如果作为MergeSort参数的表长度大于1，那么MergeSort就会将其分割。图2-30a展示了对表进行递归分割，直到每个表的长度都成1为止。然后分割的表会成对地合并起来，沿着树结构向上，直到整个表完成排序。这个过程如图2-30b所示。不过，值得注意的是，分割和合并操作是交替进行的，而不是在完成所有分割工作后再进行合并。例如，第一半表72971会在开始处理第二半表4872前被完全分割及合并。

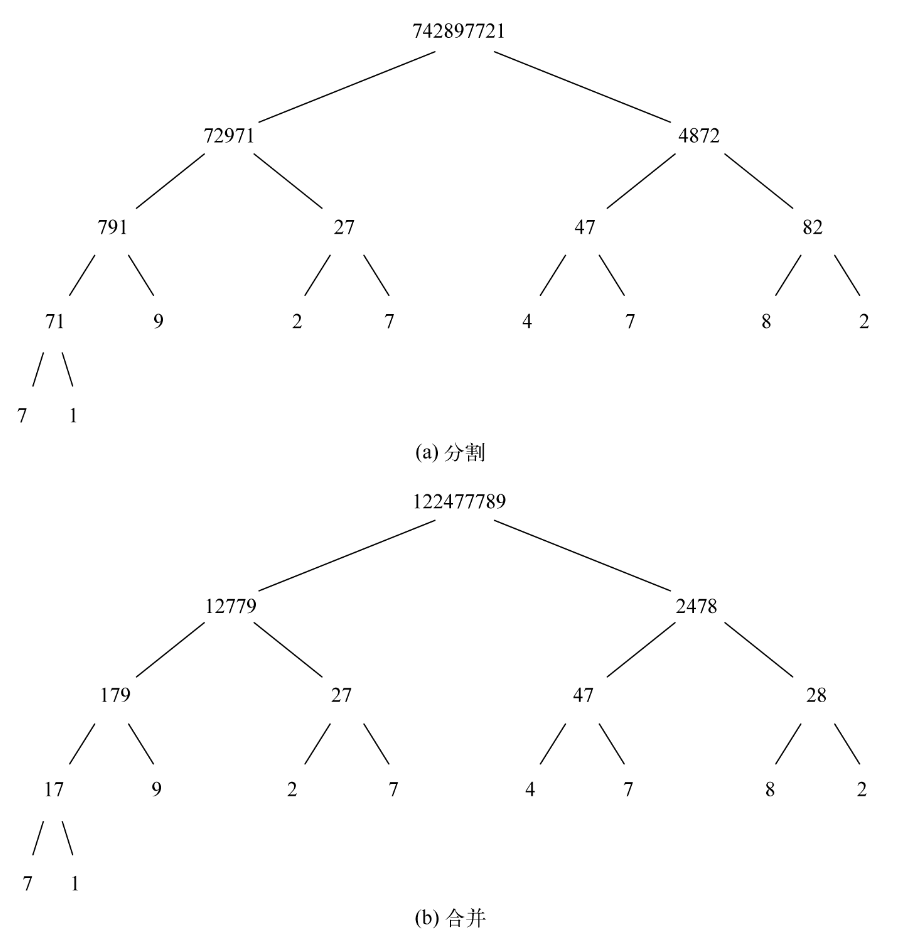


图2-30　递归的分割和合并

2.8.4 完整的程序

图2-31包含了完整的归并排序程序。它类似于图2-3中所示的选择排序程序。第(1)行中`MakeList`函数读取输入的每个整数，并通过一个简单的递归算法将其放入链表中（我们将在下一节中详细描述该递归算法）。主程序的第(2)行含有对`MergeSort`的调用，会将一个已排序表返回给`PrintList`。而`PrintList`函数会向下遍历整个已排序表，打印出每个元素。

```
     #include <stdio.h>
     #include <stdlib.h>

     typedef struct CELL *LIST;
     struct CELL {
         int element;
         LIST next;
     };

     LIST merge(LIST list1, LIST list2);
     LIST split(LIST list);
     LIST MergeSort(LIST list);
     LIST MakeList();
     void PrintList(LIST list);

     main()
     {
         LIST list;

(1)      list = MakeList();
(2)      PrintList(MergeSort(list));
     }

     LIST MakeList()
     {
         int x;
         LIST pNewCell;

(3)      if (scanf("%d", &x) == EOF) return NULL;
         else {
(4)          pNewCell = (LIST) malloc(sizeof(struct CELL));
(5)          pNewCell->next = MakeList();
(6)          pNewCell->element = x;
(7)          return pNewCell;
         }
     }

     void PrintList(LIST list)
     {
(8)      while (list != NULL) {
(9)          printf("%d\n", list->element);
(10)         list = list->next;
         }
     }
```

图2-31(a) 使用归并排序的排序程序（开头）

```
LIST MergeSort(LIST list)
{
    LIST SecondList;

    if (list == NULL) return NULL;
    else if (list->next == NULL) return list;
    else {
        SecondList = split(list);
        return merge(MergeSort(list), MergeSort(SecondList));
    }
}

LIST merge(LIST list1, LIST list2)
{
    if (list1 == NULL) return list2;
    else if (list2 == NULL) return list1;
    else if (list1->element <= list2->element) {
        list1->next = merge(list1->next, list2);
        return list1;
    }
    else {
        list2->next = merge(list1, list2->next);
        return list2;
    }
}

LIST split(LIST list)
{
    LIST pSecondCell;

    if (list == NULL) return NULL;
    else if (list->next == NULL) return NULL;
    else {
        pSecondCell = list->next;
        list->next = pSecondCell->next;
        pSecondCell->next = split(pSecondCell->next);
        return pSecondCell;
    }
}
```

图2-31(b)　使用归并排序的排序程序（结尾）

2.8.5　习题

(1) 给出对表 (1,2,3,4,5) 和 (2,4,6,8,10) 应用merge函数的结果。

(2) 假设一开始有表 (8,7,6,5,4,3,2,1)，给出产生的merge、split和Mergesort的调用序列。

(3) * 多路归并排序会将一个表均分为k个部分（或分为接近均等的k个部分），并分别为这些表排序，然后，通过比较这k个表中第一个元素的大小并选出最小的那个，将这些表合并起来。本节中描述的归并算法就是 $k=2$ 时的情况。修改图2-31中的程序，使其成为 $k=3$ 情况下的多路归并排序程序。

(4) * 重写归并排序程序，使用2.2节习题(7)中的lt和key函数，以比较任意类型的元素。

(5) 将函数merge、split和MakeList分别与图2-21联系起来。这些函数合适的大小各为多少？

2.9 证明递归程序的属性

如果想证明某个递归函数的某个特定属性，通常需要证明关于调用一次该函数的效果的命题。例如，这种效果可能是参数和返回值之间的关系，比如“调用函数，参数为i，返回$i!$”。我们经常要为函数的参数定义“大小”的概念，并通过对这个大小的归纳进行证明。可以用很多方式定义参数大小，其中包括如下内容。

(1) 某个参数的值。比如，在图2-19的阶乘递归程序中，合适的参数大小就是参数n的值。

(2) 某个参数指向的表的长度。图2-27所示的`split`递归函数就是个例子，合适的参数大小是表的长度。

(3) 参数构成的某些函数。例如，前面提到过，图2-22中递归的选择排序会对数组中有待排序的元素数目进行归纳。对参数n和i来说，该函数就是$n-i+1$。再比如，图2-24中`merge`函数合适的参数大小就是两个参数指向的表的长度之和。

不管选择了什么样的参数大小，关键是，在函数被调用时，如果参数的大小为s，那么只能以大小为$s-1$或更小的参数执行函数调用。这样我们可以对参数大小进行归纳，从而证明程序的属性。此外，当这个大小下降到某个固定的值（例如0）时，该函数一定不会再进行递归调用了，因而我们可以由依据情况开始进行归纳证明了。

✦ 示例 2.26

考虑2.7节图2-19中的阶乘程序。通过对i的归纳，证明对$i \geqslant 1$，有如下命题为真。

命题 $S(i)$。在调用`fact`时如果参数`n`的值为i，那么`fact`会返回$i!$。

依据。对$i=1$，图2-19中第(1)行的测试会使作为依据的第(2)行被执行，结果返回值为1，也就是1!。

归纳。假设$S(i)$为真，也就是说，在调用`fact`时，如果参数为某个值不小于1的i，那么它会返回$i!$。现在，考虑在调用`fact`时，变量`n`的值为$i+1$的情况。如果$i \geqslant 1$，那么$i+1$至少等于2，所以第(3)行的归纳情况是适用的，因此返回值就是$n \times fact(n-1)$。或者有，因为变量`n`的值为$i+1$，返回的结果是$(i+1) \times fact(i)$。由归纳假设，`fact(i)`返回了$i!$。因为有（$i+1$）$\times i! = (i+1)!$，所以证明了归纳步骤，也就是参数为$i+1$的`fact`函数会返回$(i+1)!$。

✦ 示例 2.27

现在，我们来看看2.8节图2-31a中的辅助例程——`MakeList`函数。该函数会创建一个用来存放输入元素的链表，并返回指向该链表的指针。我们要对输入序列中元素的数目n进行归纳，证明对$n \geqslant 0$，有以下命题为真。

命题 $S(n)$。若输入序列为x_1、x_2、…、x_n则`MakeList`会创建一个含有x_1、x_2、…、x_n的链表，并返回指向该链表的指针。

依据。依据为$n=0$，也就是，当输入序列为空时的情况。第(3)行中`MakeList`函数对`EOF`的测试会导致返回值被置为`NULL`。因此，`MakeList`正确地返回了空链表。

归纳。假设对$n \geqslant 0$，有$S(n)$为真，并考虑对有$n+1$个元素的序列调用`MakeList`时会发生的情况。假设我们刚读取了第一个元素x_1。

`MakeList`的第(4)行会创建指向新单元c的指针。由归纳假设，第(5)行会递归调用`Makelist`创建一个指针，指向存放其余n个元素x_2、x_3、…、x_{n+1}的链表。该指针在第(5)行会被

装入c的next字段。第(6)行则会将x_1装入c的element字段。第(7)行会返回第(4)行所创建的指针。该指针指向存放x_1、x_2、…、x_{n+1}这$n+1$个输入元素的链表。

这样就证明了归纳步骤，并得出MakeList能正确处理所有输入的结论。

✦　示例 2.28

在最后一个示例中，要证明图2-29中归并排序程序的正确性，其中假设split和merge函数分别能正确执行它们的任务。我们要对作为MergeSort函数的参数的表的长度进行归纳。要通过对不小于0的n进行完全归纳来证明的命题如下。

命题$S(n)$。如果list是MergeSort被调用时长度为n的表，那么MergeSort将返回具有相同元素的已排序表。

依据。要使用$S(0)$和$S(1)$作为依据。当list的长度为0时，它的值为NULL，因此图2-29中第(1)行的测试会成功，而整个函数会直接返回NULL。同样，如果list的长度是1，第(2)行的测试会成功，函数就会直接返回list。因此，MergeSort函数在n等于0或1时会返回list。这一结果证明了$S(0)$和$S(1)$，因为长度为0或1的表本来就是已排序的。

归纳。假设$n \geqslant 1$，而且对所有的$i = 0$、1、…n，都有$S(i)$为真。我们必须要证明$S(n+1)$为真。因此，要考虑长度为$n+1$的表。因为$n \geqslant 1$，所以表的长度至少为2，这样就到达了图2-29中的第(3)行。在那里，如果表的长度$n+1$为偶数，split就会把该表分为两个长度都为$(n+1)/2$的表，否则如果$n+1$为奇数，就分为长度分别为$(n/2)+1$和$n/2$的两个表。因为$n \geqslant 1$，所以这些表的长度不可能达到$n+1$。这样的话，归纳假设适用于它们，我们就可以由此得出结论，通过对第(4)行的递归调用，正确地对长度减半的表进行了排序。我们已假设merge是能正确工作的，所以返回的表也是已排序的。

习题

(1) 证明：图2-31b中的PrintList函数会打印出作为参数传入的表中的元素。需要递归证明的命题$S(i)$是什么？作为依据的i值是多少？

(2) 图2-32中的sum函数可以计算给定表中各元素之和，该表中的单元具有1.6节中的DefCell宏所定义的常见类型，这些类型在2.8节中的归并排序程序中使用过。它是通过将第一个元素加在剩余元素的和上计算所有元素之和的，而这里提到的剩余元素之和，是通过对表剩余部分递归调用该函数计算的。证明：sum函数可以正确地计算表元素之和。需要归纳证明的命题$S(i)$是什么？作为依据的i值是多少？

```
DefCell(int, CELL, LIST);

int sum(LIST L)
{
    if (L == NULL) return 0;
    else return(L->element + sum(L->next));
}

int find0(LIST L)
{
    if (L == NULL) return FALSE;
    else if (L->element == 0) return TRUE;
    else return find0(L->next);
}
```

图2-32　递归函数sum和find0

(3) 如果表中的元素至少有一个为0，那么图2-32中的`find0`函数会返回`TRUE`，否则就返回`FALSE`。如果表为空，它就返回`FALSE`，而如果第一个元素是0，就返回`TRUE`，不然的话，就对表其余部分执行递归调用，并返回为剩余部分生成的任何答案。证明：`find0`可以正确地确定表中是否出现元素0。需要归纳证明的命题 $S(i)$ 是什么？作为依据的i值是多少？

(4) * 证明：图2-24中的`merge`函数和图2-27中的`split`函数会按2.8节中所说的那样执行。

(5) 用“最少反例”直观地证明从以0和1两个值为依据开始的归纳证明是有效的。

(6) ** 证明2.7节习题(8)中用C语言实现的递归的最大公约数算法的正确性。

2.10 小结

我们从本章学习到了以下知识。

- 归纳证明、递归定义和递归程序是紧密相关的概念。它们要想“起作用”，都依赖于依据和归纳步骤。
- 在“普通归纳”或者说是“弱归纳”中，成功的那一步骤只依靠它的前一个步骤。我们经常需要进行完全归纳证明，而完全归纳中每个步骤都取决于之前的所有步骤。
- 进行排序的方法有很多。选择排序是一种简单但速度很慢的排序算法，而归并排序是一种速度比较快但比较复杂的算法。
- 归纳是证明程序或程序段能正确运转的关键。
- 分治法是一种用来设计某些优秀算法（比如归并排序）的实用技术。它的工作原理是将问题分为独立的子部分，然后将得到的结果结合起来。
- 表达式天生是由它们的操作数和运算符按照递归方式定义的。运算符可以按照它们接受参数的数量来分类：一元运算符（一个参数）、二元运算符（两个参数）以及k元运算符（k个参数）。还有，出现在两个操作数之间的二元运算符是中缀运算符，而出现在操作数之前的是前缀运算符，出现在操作数之后的则是后缀运算符。

2.11 参考文献

Roberts [1986]对递归进行了极佳的介绍。要了解更多与排序算法有关的内容，Knuth [1973]是标准的参考之作。Berlekamp [1968]讲述了从比特流中检测和收集错误的技术（2.3节介绍的是最简单的检错模式）。

Berlekamp, E. R. [1968]. *Algebraic Coding Theory*, McGraw-Hill, New York.

Knuth, D. E. [1973]. *The Art of Computer Programming*, Vol. III: *Sorting and Searching*, Addison-Wesley, Reading, Mass.

Roberts, E. [1986]. *Thinking Recursively*, Wiley, New York.

第 3 章

程序的运行时间

在第2章中，我们看到两种截然不同的排序算法：选择排序和归并排序。其实排序算法有很多种，常见的情况是通常每一个可以解决的问题都可以通过多种算法来解决。

那么，应该如何选择解决给定问题的算法呢？一般来说，应该选择易于理解、实现和记录的算法。当性能很重要时（它往往确实很重要），还需要选择能够迅速运行而且能有效使用可用计算资源的算法。因此，我们要考虑一些很微妙的问题，即如何衡量程序或算法的运行时间，以及可以采取哪些措施使程序运行得更快。

3.1　本章主要内容

本章中，我们将介绍以下主题。

- 程序性能的重要指标。
- 评估程序性能的方法。
- 大O表示法。
- 使用大O表示法估算程序的运行时间。
- 使用递推关系估算递归程序的运行时间。

3.4节和3.5节介绍的大O表示法，免去了处理那些几乎不可能确定的常量（比如常见的C语言编译器在编译某个给定的源程序时会生成的机器指令数）的麻烦，从而简化了估算程序运行时间的过程。

我们将循序渐进地介绍估算程序运行时间所需的技巧。3.6节和3.7节会展示分析不含函数调用的程序的方法。3.8节将分析具有非递归函数调用的程序。接着3.9节和3.10节会介绍如何处理递归函数。最后，3.11节将讨论递推关系的解决方案，在分析递归函数的运行时间时，对这些函数的归纳定义即称为递推关系。

3.2　算法的选择

如果需要编写的程序只是一次性处理少量数据后就弃之不用的，就应该选择自己所知的最容易实现的算法，编写并调试程序，然后就不用多管了。不过，如果需要编写在很长一段时间里由很多人使用和维护的程序，就会出现其他问题了。其一就是底层算法的可理解性，或者说是简单性。要求算法简单的原因有不少，不过最重要的也许在于，与复杂的算法相比，简单的

算法实现起来不容易出错。用简单算法实现的程序，哪怕在使用相当长一段时间后，遇到一些意外输入时曝出奇怪bug的可能性也较小。

应该将程序写得清晰明确，并仔细地记下文档，这样可便于他人维护这些程序。如果算法简单且易于理解，就更易于描述。有了好的文档，原作者之外的程序员就能方便地对原始程序加以修改（原作者经常不会做这些）；或者，如果程序完成得比较早，原作者也会对其加以修改。有很多程序员写出巧妙高效的算法后就从公司拍屁股走人了，结果后续的代码维护者只能放弃他们的算法，转而用更慢但更好理解的算法来代替，这种情况屡见不鲜。

当程序要重复运行时，它的效率以及其底层算法的效率就很重要了。我们通常会将效率与程序运行所花的时间挂钩，虽然有时程序也必须占用一些其他资源，比如：

(1) 程序变量占用的存储空间；

(2) 程序在计算机网络中产生的流量；

(3) 必须出入磁盘的数据量。

不过，对大的问题来说，对给定程序是否堪用起着决定性作用的是运作时间，而本章的主题就是运行时间。我们所要讲的程序的效率，其实就是它耗费的时间，是用程序输入大小的函数来衡量的。

通常，可理解性和效率是相互矛盾的目标。例如，比较过图2-3中的选择排序程序和图2-32中的归并排序程序的读者肯定都会认同，后者不仅更长，而且难理解得多。就算我们总结了2.2节和2.8节中给出的那些解释，在程序中添加了经过深思熟虑的注释，结果依然如此。不过，也正如我们将要了解的，只要待排序的元素个数过百，归并排序的效率就会比选择排序的效率高得多。不巧的是，这种情况太普遍了——对大数据量来说有效率的算法，编写和理解起来往往比那些相对低效的算法更加复杂。

算法的可理解性，或者说是简单性，是有些主观的概念。我们可以在某种程度上克服算法不够简单的问题，即在注释和程序文档中对算法进行到位的解释。编写文档的人始终要考虑阅读这些代码及其注释的人：一般人能明白这是在说什么吗？是否需要进一步的解释、细节、定义和示例？

另一方面，程序的效率是个客观的问题：程序所花的时间就是那么多，没什么争议的余地。不过我们没办法用所有可能的（通常是无数的）输入来运行程序。因此，我们要对程序运行时间加以度量，因为它总结了程序处理所有输入的性能，通常是用一个诸如“n^2”这样的简单表达式来度量的。本章下面几节的主题就是如何度量程序的运行时间。

3.3　度量运行时间

一旦我们认同可以通过度量程序的运行时间对程序加以评估，就要面对确定实际运行时间的问题。总结运算时间的两种主要方法是：

(1) 基准测试；

(2) 分析。

我们将依次介绍这两种方法，不过本章主要讲的还是用于分析程序或算法的技术。

3.3.1　基准测试

在比较用于完成相同任务的两个或多个程序时，制定一小组可用作基准的典型输入是一种

惯例。也就是说，我们愿意接受基准输入作为这些任务组合的代表，并假设能顺利处理基准输入的程序能顺利处理所有输入。

例如，评估排序算法的基准可能包含一小组数字（比如圆周率的前20位数字）、一个中等规模的输入组（比如得克萨斯州的邮政编码集合），以及一个大规模输入组（比如布鲁克林区电话目录中的电话号码集合）。我们可能还想知道，在对空集、单元素集以及已排序表排序时，程序是否能有效及正确地工作。有趣的是，有些排序算法在处理已排序表时的性能惨不忍睹。[①]

90-10法则

与基准测试一样，确定要分析的程序在哪里花了时间通常也是很实用的。这种评估程序性能的方法称为剖析（profiling），而且多数程序设计环境都包含有剖析器（profiler）这种工具，会为程序中每条语句关联一个表示执行这条语句所花时间的数字。还有一种相关的实用程序，名叫语句计数器，用于确定对于给定的输入集，源程序中每条语句执行的次数。

很多程序都具有这样的特性，即大部分运行时间都花在一小部分源代码上了。有这么一条非正式的法则：90%的运行时间花在了10%的代码上。尽管准确的百分比是视程序而定的，不过“90-10法则”还是表明了多数程序中运行时间主要花在了哪里。想要加快程序运行速度，最简单的一种方法就是对程序加以剖析，并对程序“热点”（也就是程序中花掉大部分运行时间的部分）的代码加以改进。例如，我们在第2章中提到过，用等价的迭代函数替代递归函数是可能为程序提速的。不过，这种做法只有在递归函数正好是程序中占用大部分运行时间的部分时才奏效。

在极端情况下，即便我们将只占用10%时间的那90%的代码所花的时间变为0，程序总的运行时间也只减少了10%。然而，如果将10%的程序所占用的那90%的时间减半，总运行时间就将减少45%。

3.3.2　对程序的分析

要分析程序，首先要按大小为输入分组。正如在2.9节中与证明递归程序属性一起讨论的那样，用来表示输入大小的度量是因程序而异的。对排序程序来说，待排序元素的数量就是个很不错的度量。对于求解n元线性方程组的程序，拿n作为问题的大小是很平常的。其他的程序可能使用某个特定输入的值作为程序输入的表的长度，或作为输入的数组的大小，或是诸如此类的度量的组合。

3.3.3　运行时间

用函数$T(n)$来表示程序或算法处理大小为n的任意输入所花的时间是很方便的。我们将$T(n)$称为程序的运行时间。例如，某个程序的运行时间可能是$T(n)=cn$，其中c是某个常数。换个说法就是，该程序的运行时间，与其要处理的输入的大小是线性相关的。这样的程序或算法就是线性时间的，或者直接说成是线性的。

---

① 选择排序和归并排序都不在此列，它们在处理有序列表时所花的时间几乎与为相同长度的任一列表排序所花的时间相同。

我们可以将运行时间$T(n)$看作程序执行的C语言语句的数量，或是在某标准计算机上运行程序所花的时长。在多数情况下，我们都不会明确指出$T(n)$的具体单位。事实上，正如我们在下一节中将要看到的，在谈论程序的运行时间时，可以只用某个（未知的）常数因子乘上$T(n)$来表示。

很多时候，程序的运行时间取决于某个特定的输入，而不仅仅取决于输入的大小。在这类情况下，我们将$T(n)$定义为*最坏情况*运行时间，也就是所有大小为n的输入所能造成的最大运行时间。

另一种常见的性能度量是$T_{avg}(n)$，即程序处理所有大小为n的输入的平均运行时间。平均运行时间有时候是对实际性能更为现实的反映，不过它往往比最坏情况运行时间更难计算。"平均运行时间"中"平均"的概念还意味着，所有大小为n的输入是等可能性的，而这在某个给定情况下既可能为真，也可能不为真。

✦ 示例 3.1

让我们估算一下图3-1中所示的`SelectionSort`程序段的运行时间。这些语句的编号与图2-2中的编号如出一辙。这段代码的目的是要将数组A从`A[i]`到`A[n-i]`这部分中最小元素的下标赋值给`small`。

```
(2)         small = i;
(3)         for(j = i+1; j < n; j++)
(4)             if (A[j] < A[small])
(5)                 small = j;
```

图3-1 选择排序的内层循环

一开始，我们需要对时间单位加以简单定义。后面我们会详细介绍这一问题，不过在这里，以下简单模式是有效的。可以将每次执行赋值语句记作一个时间单位。在第(3)行，要为`for`循环开头`j`的初始化记上一个时间单位，为测试是否有$j<n$记上一个单位，并为减少`j`记上一个单位，每次循环皆是如此。最后，每执行一次第(4)行的测试，就要记上一个单位。

首先，让我们考虑一下内层循环的循环体：第(4)行和第(5)行。第(4)行的测试总是要执行的，不过第(5)行的赋值只有在测试成功的情况下才执行。因此，该循环体会消耗1到2个时间单位，这取决于数组`A`中的数据。如果要采纳最坏情况，就可以假设循环体要消耗2个时间单位。我们会进行$n-i-1$次`for`循环，而每进行一次循环都要执行一遍循环体（2个时间单位），接着递增`j`并测试$j<n$（又是2个时间单位）。因此，进行循环所花的时间单位是$4(n-i-1)$。除了这个数字之外，我们还要加上第(2)行初始化`small`的1个时间单位，第(3)行初始化`j`的1个时间单位，以及在第(3)行第一次测试$j<n$的1个时间单位（这与循环的任一次迭代的终止无关）。因此，图3-1中的程序段的总运行时间为$4(n-i)-1$。

将图3-1所影响数据的"大小"m指定为$m=n-i$是很自然的，因为这是它所影响的数组`A[i..n-1]`的长度。那么运行时间$4(n-i)-1$就可以表示为$4m-1$。因此，图3-1的运行时间$T(m)$就是$4m-1$。

3.3.4 不同运行时间的比较

假设对某个问题，可以选择使用运行时间为$T_A(n)=100n$的线性时间程序A，以及运行时间

为 $T_B(n)=2n^2$ 的二次幂时间程序B。假设这两个运行时间是在同一特定计算机上处理大小为n的输入所花的毫秒数。[①]运行时间图见图3-2。

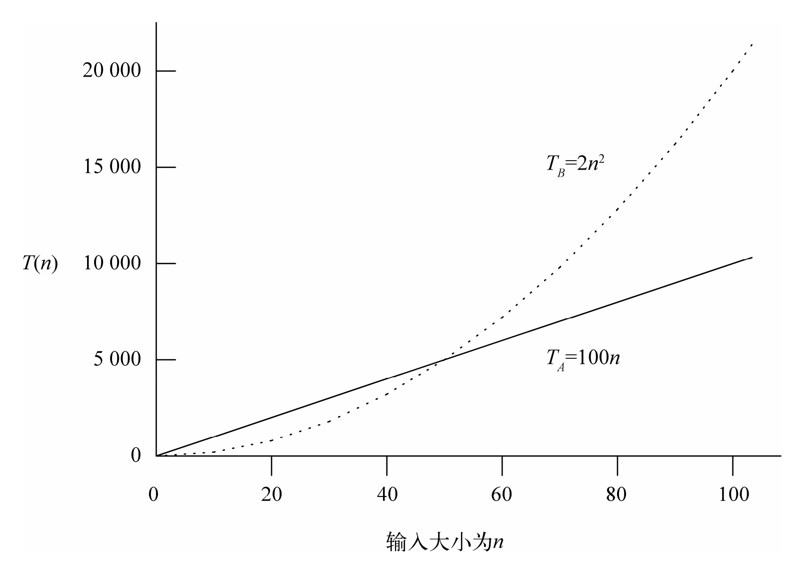


图3-2　线性程序与二次幂程序的运行时间

从图3-2可知，对大小小于50的输入来说，程序B要比程序A快。当输入的大小大于50时，程序A就要更快了，而且从50这个临界点开始，输入越大，程序A相比程序B而言优势就越大。对大小为100的输入，A要比B快上2倍，而对大小为1000的输入，A要快上20倍。

程序运行时间的函数形式最终确定了我们能用该程序解决多大的问题。随着计算机速度的不断变快，与运行时间增长迅速的程序相比，那些运行时间增长缓慢的程序在可处理问题的规模上能取得更大的提高。

再次假设图3-2所示的程序运行时间是以毫秒计的，图3-3中的表格表示了在同一台计算机上，花同样的时间，使用两种程序分别能解决多大的问题。例如，假设可以接受100秒的计算时间。如果计算机的速度加快10倍，那么在100秒内能处理之前需要花1000秒去处理的问题。对算法A来说，我们现在可以解决10倍大小的问题，而对算法B来说，只可以解决3倍大小的问题。因此，随着计算机速度的持续加快，通过使用低增长率的算法和程序可以获得更为显著的优势。

时间（秒）	使用程序A可解决的最大问题的大小	使用程序A可解决的最小问题的大小
1	10	22
10	100	70
100	1000	223
1000	10000	707

图3-3　在可用时间段函数可解决问题的大小

① 这里A和B的关系，与归并排序和选择排序的关系没有太多不同。我们在3.10节中将会看到，归并排序的运行时间是以$n\log n$的速度增长的。

别在乎算法的效率，再等上几年就行了

大家可能经常听到这样的说法、不需要缩短算法的运行时间或是选择更高效的算法，因为计算机的速度每隔几年就会翻番，而且不需要多久，任何算法，不管有多低效，所花的时间都会少到没有人在意了。这一论调的出现已经有几十年的时间了，但计算资源需求上限尚未出现，因此，我们一般都不接受硬件改善可以让高效算法的研究变成无用功这种观点。

不过，也存在我们不需要过分考虑效率的情况。例如，某所学校在每学期期末都要将存储在某台计算机中的学生电子成绩表打印成纸质成绩单。该操作所花的时间大概与要报告的成绩数量成线性关系，就像假想算法*A*那样。如果学校更换了一台速度快上10倍的计算机，完成这项工作所花的时间就会变为原来的十分之一。不过，学校因此要扩招10倍，或是要求每个学生增加10倍的课程，这是很不现实的。计算机的提速不会影响到成绩单程序的输入大小，因为这一大小是受其他因素限制的。

另一方面，还会存在另外一些问题，我们凭借新兴的计算资源有了一些解决的头绪，不过它们的“大小”却超出了现有技术的处理能力。这样的问题包括自然语言的理解、计算机视觉（对数字化图像的理解），以及各种对人机“智能”交互的尝试。不管是通过改善算法还是通过提升机器性能，所获得的加速都将提升我们在接下来几年里处理这些问题的能力。此外，当它们变成“简单”的问题后，我们现在很难想象的新一代挑战又会替代它们摆在计算机面前。

3.3.5 习题

(1) 考虑一下图2-13中的阶乘程序段，设输入大小为读取的n的值。每次执行赋值、读和写语句记为一个时间单位，每进行一次`while`循环条件测试记为一个时间单位，计算该程序的运行时间。

(2) 为2.5节习题(1)以及图2-14中的程序段给出恰当的输入大小。运用上一题中的计数规则，确定这两个程序的运行时间。

(3) 假设程序A花费 $2^n/1000$ 个时间单位，程序B花费 $1000n^2$ 个时间单位。对哪些n值来说，程序A花的时间比程序B少？

(4) 对上一题中的两个程序，在10^6个、10^9个和10^{12}个时间单位内能解决的问题各有多大？

(5) 假设程序A花费 $1000n^4$ 个时间单位，程序B花费 n^{10} 个时间单位，重复习题(3)和习题(4)中的练习。

3.4 大 O 运行时间和近似运行时间

假设我们编写了一个C语言程序，并选择了想要它处理的特定输入。程序处理这一输入的运行时间仍取决于以下两个因素。

(1) 运行该程序的计算机。一些计算机执行指令的速度比其他计算机更快，最快的超级计算机与最慢的个人计算机之间的性能比远大于1000：1。

(2) 生成计算机可执行程序所使用的特定C语言编译器。在同一计算机上，执行不同程序所用的时间是不一样的，即便这些程序有着相同的功效。

这样一来，我们就不能看着C语言程序及其输入，然后判断说：“这个任务要花上3.21秒。”除非知道用的什么计算机和编译器。此外，就算我们知道程序、输入、机器和编译器，要准确预计将要执行的机器指令数通常也是一项过于复杂的任务。

出于这些原因，我们通常用大O表示法来表示程序的运行时间，该方法让我们可以不去考虑如下常数因子。

(1) 特定编译器生成机器指令的平均数。

(2) 特定计算机每秒执行机器指令的平均数。

例如，就像在示例3.1中那样，我们研究的`SelectionSort`程序段处理长度为m的数组将耗时$4m-1$。不过这里我们不这么说，而是说它耗时$O(m)$，非正式的含义是“某个常数乘以m”。

“某个常数乘以m”这一表述不仅能让我们忽略那些与编译器和计算机相关的未知常数，还让我们可以作出一些起简化作用的假设。例如，在示例3.1中，假设所有的赋值语句会消耗长短相同的一段时间，而在测试`for`循环的终止、随着循环进行递增`j`，以及进行变量初始化等工作时，也都会消耗这样长的一段时间。因为这些假设在实际情况中都是不可能的，所以在运行时间方程$T(m)=4m-1$中，常数4和−1是对事实的最佳逼近。可以更近似地将$T(m)$描述为“某个常数乘以m，再加上或减去某个常数”，甚至描述为“最多与m成正比”。$O(m)$表示法使我们可以在不涉及不可知或无意义常数的情况下作出这些陈述。

另一方面，将程序段的运行时间表示为$O(m)$，也告诉我们一些非常重要的事情。它表明，执行处理逐步变大的数组的程序，所花的时间是线性增长的，就像3.3节末尾的图3-2和图3-3中假想的程序A那样。因此，该程序段表示的算法，优于运行时间增长更快的算法（比如在上文的讨论中与程序A相对比的假想程序B）。

3.4.1　大O的定义

我们现在要给出某个函数是另一个函数的“大O”的正式定义。设有函数$T(n)$，这通常是某个程序的运行时间，以输入大小为n的函数来度量。要让函数适用于度量程序的运行时间，我们假设有：

(1) 参数n被限定为非负整数；

(2) 值$T(n)$对所有的参数n来说都非负。

设$f(n)$是某个定义在非负整数n之上的函数。如果除了对某些较小的n值之外，$T(n)$至多是某个常数乘以$f(n)$，我们就可以说

“$T(n)$是$O\big(f(n)\big)$”。

正式地说，如果存在某个整数n_0以及某个大于0的常数c，使得对所有大于n_0的整数n，都有$T(n)\leqslant cf(n)$，那么我们就说$T(n)$是$O\big(f(n)\big)$。

我们把数对n_0和c称为“$T(n)$是$O\big(f(n)\big)$”这一事实的*证物*（witness）。在接下来的证明中，该证物可以为$T(n)$和$f(n)$的大O关系“作证”。

3.4.2　证明大O关系

可以应用“大O”的定义证明对特定的函数T和f，$T(n)$就是$O\big(f(n)\big)$。我们会通过选择特定的证物n_0和c，接着证明$T(n)\leqslant cf(n)$，从而完成这一证明。证明过程必须假设n是非负整数，且不小于我们选择的n_0。通常，证明过程涉及一些代数和不等式的变换。

✦　示例 3.2

假设有某个程序，其运行时间为$T(0)=1$，$T(1)=4$，$T(2)=9$，一般表示为$T(n)=(n+1)^2$。

我们就可以说$T(n)$是$O(n^2)$，或者说$T(n)$是二次幂的，因为可以选择证物$n_0=1$和$c=4$。然后需要证明$(n+1)^2 \leqslant 4n^2$，其中有$n \geqslant 1$。在证明过程中，我们将表达式$(n+1)^2$展开为n^2+2n+1。只要$n \geqslant 1$，我们就知道有$n \leqslant n^2$而且有$1 \leqslant n^2$。因此

$$n^2+2n+1 \leqslant n^2+2n^2+n^2=4n^2$$

此外，也可以选择证物$n_0=3$和$c=2$，因为，正如大家可以验证的，对所有的$n \geqslant 3$，都有$(n+1)^2 \leqslant 2n^2$。

不过，不能选择$n_0=0$，因为若$n_0=0$，那么对$n=0$，我们可以证明$(0+1)^2 \leqslant c0^2$，也就是有1小于等于c乘以0。因为不管选择什么c，都有$c \times 0=0$，而$1 \leqslant 0$是不成立的，所以如果我们选择$n_0=0$就玩完了。不过没关系，因为要证明$(n+1)^2$是$O(n^2)$，所以只要找出一组可行的证物n_0和c就行了。

大O证明的模版

请记住：所有的大O证明基本遵循相同的形式，只有代数变换是各异的。要证明$T(n)$就是$O\big(f(n)\big)$，要做的只有下面两件事。

(1) 说明证物n_0和c。这些证物必须是特定的常数，比如$n_0=47$和$c=12.5$。还有，n_0必须是非负整数，而c必须是正实数。

(2) 通过适当的代数变换，证明对所选择的特定证物n_0和c，如果$n \geqslant n_0$，则有$T(n) \leqslant cf(n)$。

这可能看起来有些奇怪，虽然$(n+1)^2$大于n^2，但是我们还是说$(n+1)^2$是$O(n^2)$。其实，也可以说$(n+1)^2$是任意分之n^2的大O，例如$O(n^2/100)$。要看看原因的话，选择证物$n_0=1$和$c=400$。那么如果$n \geqslant 1$，由与示例3.2中一样的推导可知

$$(n+1)^2 \leqslant 400(n^2/100)=4n^2$$

这些现象背后的基本原则如下。

(1) 常数因子不产生影响。对于任意正值常数d和任意函数$T(n)$，$T(n)$是$O\big(dT(n)\big)$，不论d是很大的数，还是很小的分数，只要$d>0$即可。要知道为什么，可以选择证物$n_0=0$和$c=1/d$。[①]那么就有$T(n) \leqslant c\big(dT(n)\big)$，因为$cd=1$。类似地，若已知$T(n)$是$O\big(f(n)\big)$，便也知道对任何的$d>0$，有$T(n)$是$O\big(df(n)\big)$，即便$d$很小。因为我们知道，对某个常数$c_1$和所有的$n \geqslant n_0$，有$T(n) \leqslant c_1 f(n)$。如果选择$c=c_1/d$，那么就可以看到，对$n \geqslant n_0$，有$T(n) \leqslant c\big(df(n)\big)$。

(2) 低阶项不产生影响。假设$T(n)$是形如

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

的多项式，其中开头的系数a_k为正数。然后我们可以扔掉除第一项（就是具有最高指数k的那项）之外的所有项，并利用规则(1)，忽略常数a_k，直接用1代替它。也就是说，我们可以得出$T(n)$就是$O(n^k)$。为了证明这一点，设n_0=1，并设c是各系数a_i中所有正系数的和，其中$0 \leqslant i \leqslant k$。如果系数$a_j$是0或负数，那么肯定有$a_j n^j \leqslant 0$。如果$a_j$为正，那么对所有的$j<k$，只要$n \geqslant 1$，

---

① 请注意，虽然要求选择常数而不是函数作为证物，但选择$c=1/d$是没有错的，因为d本身也是个常数。

都有 $a_j n^j \leqslant a_k n^k$ 。因此 $T(n)$ 不大于 n^k 乘以所有正系数之和，或者说是 $T(n) \leqslant cn^k$ 。

关于大O的谬论

对“大O”的定义是很诡异的，在该定义中，在检查完 $T(n)$ 和 $f(n)$ 后，我们要一次性选择证物 n_0 和c，接着证明对所有的 $n \geqslant n_0$ ，都有 $T(n) \leqslant cf(n)$ 。不能为每个n值重新选择c和（或）n_0 。例如，大家可能偶尔会看到如下证明 n^2 为 $O(n)$ 的谬误“证明”。“选择 $n_0 = 0$ ，并为每个n选择 $c = n$ 。然后有 $n^2 \leqslant cn$ 。”这种论述是无效的，因为我们要求在不知道n的情况下一次性选定c。

✦ 示例 3.3

作为规则(1)（“常数因子不产生影响”）的例子，我们看到$2n^3$是$O(0.001n^3)$。令 $n_0 = 0$ ，而且 $c = 2/0.001 = 2000$ 。那么显然有，对所有的 $n \geqslant 0$ ， $2n^3 \leqslant 2000(0.001n^3) = 2n^3$ 。

作为规则(2)（“低阶项不产生影响”）的例子，考虑多项式 $T(n) = 3n^5 + 10n^4 - 4n^3 + n + 1$ 。最高位的项是 n^5 ，我们就说 $T(n)$ 是 $O(n^5)$ 。要验证该说法，令 $n_0 = 1$ ，c等于所有正系数的和。正系数的项包含指数为5、4、1和0的这些项，其系数分别为3、10、1和1。因此，令 $c = 15$ 。我们可以说，对 $n \geqslant 1$ ，有

$$3n^5 + 10n^4 - 4n^3 + n + 1 \leqslant 3n^5 + 10n^5 + n^5 + n^5 = 15n^5 \tag{3.1}$$

我们可以通过对正系数项的匹配来验证不等式(3.1)，也就是， $3n^5 \leqslant 3n^5$ ， $10n^4 \leqslant 10n^5$ ， $n \leqslant n^5$ ，以及 $1 \leqslant n^5$ 。而且，因为 $-4n^3 \leqslant 0$ （之前假设n为正数），所以可以忽略不等式(3.1)左边的负系数项。因此，不等式(3.1)的左边，也就是 $T(n)$ ，要小于等于不等式的右边，也就是 $15n^5$ ，或者说是 cn^5 。由此可以得出 $T(n)$ 是 $O(n^5)$ 的结论。

其实，低阶项可以删除的原则不仅适用于多项式，而且适用于任何表达式之和。也就是说，如果随着n趋近无穷大， $h(n)/g(n)$ 的比值趋近于0，则可以说 $h(n)$ 比 $g(n)$ “增长得慢”，或者说 $h(n)$ 的“增长率低于” $g(n)$ ，这样就可以忽略 $h(n)$ 。也就是说 $h(n) + g(n)$ 是 $O\big(g(n)\big)$ 。

例如，设 $T(n) = 2^n + n^3$ 。众所周知，多项式（比如 n^3 ）要比指数式（比如 2^n ）增长得慢。因为随着n的增大， $n^3/2^n$ 趋近于0，所以我们可以扔掉低阶项，并得出 $T(n)$ 是 $O(2^n)$ 的结论。

要正式地证明 $2^n + n^3$ 是 $O(2^n)$ ，令 $n_0 = 10$ ， $c = 2$ 。必须证明，对 $n \geqslant 10$ ，有

$$2^n + n^3 \leqslant 2 \times 2^n$$

如果从两边都减去 2^n ，就会发现这是要证明对 $n \geqslant 10$ ，有 $n^3 \leqslant 2^n$ 。

对 $n = 10$ ，我们有 $2^{10} = 1024$ 。而 $10^3 = 1000$ ，因此对 $n = 10$ ，有 $n^3 \leqslant 2^n$ 。n每增加1， 2^n 就会翻倍，而 n^3 则是会乘以 $(n+1)^3/n^3$ 这个量，而当 $n \geqslant 10$ 时，这个量是小于2的。因此，随着n的增大，n^3 会逐步小于 2^n 。我们可以得出结论：对 $n \geqslant 10$ ，有 $n^3 \leqslant 2^n$ ，因此有 $2^n + n^3$ 是 $O(2^n)$ 。

3.4.3　证明大O关系不成立

如果两个函数之间的大O关系成立，就可以通过找出证物来证明这种关系。然而，如果某个函数 $T(n)$ 不是另一个函数 $f(n)$ 的大O呢？答案就是，经常可以证明某个特定的函数 $T(n)$ 不是 $O\big(f(n)\big)$ 。证明方法是，假设证物 n_0 和c存在，并推理出矛盾。下面要介绍这种证明的一个例子。

✦ 示例 3.4

在前文附注栏“关于大O的谬论”中，我们声称n^2不是$O(n)$。我们可以证明该声明，方法如下。假设n^2是$O(n)$，然后就存在证物n_0和c，使得对所有的$n \geqslant n_0$，都有$n^2 \leqslant cn$。不过如果我们选择n_1等于$2c$和n_0中较大者，就会有不等式

$$(n_1)^2 \leqslant cn_1 \tag{3.2}$$

一定成立（因为$n_1 \geqslant n_0$，而且对所有的$n \geqslant n_0$，$n^2 \leqslant cn$都是成立的）。

如果将不等式(3.2)两边都除以n_1，就有$n_1 \leqslant c$。然而，我们还选择了n_1至少是$2c$。因为证物c一定为正数，所以n_1不可能既小于c又大于$2c$。因此，可以证明“n^2是$O(n)$”的证物n_0和c不存在，由此可以得出结论：n^2不是$O(n)$。

3.4.4 习题

(1) 考虑以下4个函数。

$f_1 : n^2$

$f_2 : n^3$

$$f_3 : \begin{cases} n^2，n\text{为奇数} \\ n^3, n\ \text{为偶数} \end{cases}$$

$$f_4 : \begin{cases} n^2，n\text{ 为质数} \\ n^3，n\text{ 为合数} \end{cases}$$

对等于1、2、3、4的i和j，分别确定$f_i(n)$是不是$O\big(f_j(n)\big)$。要么给出证明大O关系的n_0和c的值，要么假设存在这样的n_0和c，并推理出矛盾，证明$f_i(n)$不是$O\big(f_j(n)\big)$。提示：请记住，除了2之外，所有的质数都是奇数。还要记住，质数有无数个，而合数也有无数个。

(2) 有以下一些大O关系。请为每个大O关系给出可用来证明这种关系的证物n_0和c。选择最小的一组证物，也就是说n_0-1和c不是证物，而如果$d<c$，那么n_0和d也不是证物。

(a) n^2是$O(0.001n^3)$。

(b) $25n^4-19n^3+13n^2-106n+77$是$O(n^4)$。

(c) 2^{n+10}是$O(2^n)$。

(d) n^{10}是$O(3^n)$。

(e)* $\log_2 n$是$O(\sqrt{n})$。

(3) * 证明：如果对所有的n有$f(n)<g(n)$，那么$f(n)+g(n)$是$O\big(g(n)\big)$。

(4) ** 假设$f(n)$是$O\big(g(n)\big)$，而且$g(n)$是$O\big(f(n)\big)$。那么$f(n)$和$g(n)$之间有什么关系？是不是一定有$f(n)=g(n)$？随着n趋近无穷大，$f(n)/g(n)$的极限是否一定存在？

证明大O关系不成立的模板

证明函数$T(n)$不是$O\big(f(n)\big)$的常见证明过程如下。示例3.4就展示了这样的证明过程。

(1) 首先假设存在证物n_0和c，使得对所有的$n \geqslant n_0$，都有$f(n) \leqslant cg(n)$。这里，n_0和c是表示未知证物的符号。

(2) 定义特定的整数n_1，用与n_0和c相关的形式表示（例如，在示例3.4中，我们选择的是$n_1 = \max(n_0, 2c)$）。该n_1是用来证明$T(n_1) \leqslant cf(n_1)$的n的值。

(3) 证明，对于这个选定的 n_1，有 $n_1 \geqslant n_0$。这一部分是非常简单的，因为我们在第(2)步中选择的 n_1 至少是 n_0。

(4) 声明因为有 $n_1 \geqslant n_0$，所以一定有 $T(n_1) \leqslant cf(n_1)$。

(5) 通过证明对我们选择的这个 n_1 有 $T(n_1) > cf(n_1)$，从而推导出矛盾。选择与c有关的 n_1 可以让这个部分变得很简单，就像示例3.4中所做的那样。

3.5　简化大 O 表达式

正如我们在3.4节中看到的，通过舍弃一些常数因子和低阶项，可以简化大O表达式。我们将会看到，在分析程序时，作出这样的简化有多重要。一般来说，某个程序的运行时间来源于程序中很多不同的语句或程序段，而一小部分程序占用大量运行时间的情况也很平常（由“90-10”法则可知）。通过舍弃一些低阶项，并将相等或近似相等的项结合起来，通常能大大简化表示运行时间的大O表达式。

3.5.1　大O表达式的传递律

首先，我们要拿出考虑大O表达式时的一个实用规则。诸如≤这样的关系，就被称为传递的，因为它遵循“若 $A \leqslant B$，且 $B \leqslant C$，则 $A \leqslant C$”这样的法则。例如，因为 $3 \leqslant 5$，$5 \leqslant 10$，所以我们可以确定 $3 \leqslant 10$。

而“是f的大O”这样的关系是另一种具有传递性的关系。也就是说，如果 $f(n)$ 是 $O\big(g(n)\big)$，而且 $g(n)$ 是 $O\big(h(n)\big)$，就有 $f(n)$ 是 $O\big(h(n)\big)$。要知道原因，首先假设 $f(n)$ 是 $O\big(g(n)\big)$。那么存在证物 n_1 和 c_1，使得对所有的 $n \geqslant n_1$，都有 $f(n) \leqslant c_1 g(n)$。类似地，如果 $g(n)$ 是 $O\big(h(n)\big)$，就存在证物 n_2 和 c_2，使得对所有的 $n \geqslant n_2$，都有 $g(n) \leqslant c_2 h(n)$。

多项和指数大O表达式

多项式的次数是指多项式所有项中的最高指数。例如，示例3.3和示例3.5中提到的多项式 $T(n)$ 的次数为5，因为其最高阶项为 $3n^5$。从我们已经阐明的两个原则（常数因子不产生影响，以及低阶项不产生影响），以及大O表达式的传递律，可知以下几点。

(1) 如果 $p(n)$ 和 $q(n)$ 都是多项式，且 $q(n)$ 的次数大于等于 $p(n)$ 的次数，就有 $p(n)$ 是 $O\big(q(n)\big)$。

(2) 如果 $q(n)$ 的次数小于 $p(n)$ 的次数，那么 $p(n)$ 不是 $O\big(q(n)\big)$。

(3) 指数式是指形如 a^n 的表达式（其中 $a>1$）。指数式要比多项式增长得更快。也就是说，我们可以为任一多项式 $p(n)$ 证明，$p(n)$ 是 $O\big(a^n\big)$。例如，n^5 是 $O\big((1.01)^n\big)$。

(4) 反过来，对 $a>1$，不存在指数式 a^n 为多项式 $p(n)$ 的 $O\big(p(n)\big)$。

设 n_0 是 n_1 和 n_2 二者中的较大值，而且令 $c = c_1 c_2$。我们声称 n_0 和c为“$f(n)$ 是 $O\big(h(n)\big)$”这

一事实的证物。这里假设 $n \geqslant n_0$ 。因为 $n_0 = \max(n_1, n_2)$ ，所以我们知道 $n \geqslant n_1$ 且 $n \geqslant n_2$ 。因此，$f(n) \leqslant c_1 g(n)$ ，且 $g(n) \leqslant c_2 h(n)$ 。

现在用 $c_2 h(n)$ 替换不等式 $f(n) \leqslant c_1 g(n)$ 中的 $g(n)$ ，就证明了 $f(n) \leqslant c_1 c_2 h(n)$ 。该不等式就证明了 $f(n)$ 是 $O\big(h(n)\big)$ 。

✦ 示例 3.5

从示例3.3中可知

$$T(n) = 3n^5 + 10n^4 - 4n^3 + n + 1$$

是 $O(n^5)$ ，还可以从规则“常数因子不产生影响”中知道 n^5 是 $O(0.01n^5)$ 。通过大O的传递律，可知 $T(n)$ 是 $O(0.01n^5)$ 。

3.5.2 描述程序的运行时间

我们之前对程序的运行时间 $T(n)$ 的定义是，程序处理大小为n的任意输入所耗费时间单位的最大值。我们还说过，要确定 $T(n)$ 的准确公式，就算不是不可能，也将非常困难。通常，可以用大O表达式 $O\big(f(n)\big)$ 作为 $T(n)$ 的上限，从而将问题大大简化。

例如，`SelectionSort`程序的运行时间 $T(n)$ 的上限是 an^2 ，其中a是某个常数，而且 $n \geqslant 1$ ，我们将在3.6节中展示这一事实。然后可以说`SelectionSort`的运行时间是 $O(n^2)$ 。从直觉上讲，这一陈述是最为实用的，因为 n^2 是个非常简单的函数，而且有关其他简单函数的更强陈述（比如“$T(n)$ 是 $O(n)$”）都为假。

不过，因为大O表示法的本性，还可以说运行时间 $T(n)$ 是 $O(0.01n^2)$ ，或 $O(7n^2 - 4n + 26)$ ，或者是任何二次多项式的大O。原因在于，n^2 是任意二次式的大O，而根据传递律，就可以从 $T(n)$ 是 $O(n^2)$ 这一事实得出 $T(n)$ 是任意二次式的大O。

更糟的是，n^2 还是任意三次或更高次多项式，或者是任意指数式的大O。因此，再次利用传递性，$T(n)$ 是 $O(n^3)$ ，$O(2^n + n^4)$ ，等等。不过我们将会解释，为什么 $O(n^2)$ 是表示`SelectionSort`程序的运行时间的首选。

3.5.3 紧凑性

首先，我们一般都想要达到可以证明的“最紧”大O上界。也就是说，如果 $T(n)$ 是 $O(n^2)$ ，我们就想作出这一表述，而不是作出“$T(n)$ 是 $O(n^3)$”这种技术上正确但更弱的表述。另一方面，这种方式又存在某种疯狂性，因为如果我们喜欢用 $O(n^2)$ 作为运行时间的表达式，就应该更喜欢 $O(0.5n^2)$ ，因为它更“紧凑”，而对 $O(0.01n^2)$ 的喜爱应该就更甚了。不过，因为在大O表达式中，常数因子是不产生影响的，所以通过缩小常数因子让预估运行时间“更紧凑”的尝试是没有意义的。因此，只要有可能，我们就会试着使用常数因子为1的大O表达式。

图3-4列出了一些比较常见的程序运行时间，以及它们的非正式名称。特别要注意，$O(1)$ 是表示“某个常数”的惯用简写形式，而且我们还将反复使用这种用意的 $O(1)$ 。

大O	非正式名称
$O(1)$	常数
$O(\log n)$	对数
$O(n)$	线性
$O(n\log n)$	$n\log n$
$O(n^2)$	二次
$O(n^3)$	三次
$O(2^n)$	指数

图3-4　一些常见大O运行时间的非正式名称

更精确地讲，如果同时满足如下两点

(1) $T(n)$ 是 $O\big(f(n)\big)$；

(2) 如果 $T(n)$ 是 $O\big(g(n)\big)$，那么 $f(n)$ 是 $O\big(g(n)\big)$ 也为真（通俗地讲，我们找不出这样一个函数 $g(n)$，它至少与 $T(n)$ 增长得一样快，却又比 $f(n)$ 增长得慢）。

那么我们就说 $f(n)$ 是 $T(n)$ 的紧大O边界（tight big-oh bound）。

✦ 示例 3.6

设 $T(n)=2n^2+3n$，而且 $f(n)=n^2$。我们说，$f(n)$ 是 $T(n)$ 的紧边界（tight bound）。要知道为什么，先假设 $T(n)$ 是 $O\big(g(n)\big)$。然后，存在常数 c 和 n_0，使得对所有的 $n\geqslant n_0$，有 $T(n)=2n^2+3n\leqslant cg(n)$。那么对 $n\geqslant n_0$，有 $g(n)\geqslant(2/c)n^2$。因为 $f(n)$ 是 n^2，所以可得出，对 $n\geqslant n_0$，有 $f(n)\leqslant(c/2)g(n)$。因此，$f(n)$ 是 $O\big(g(n)\big)$。

另一方面，$f(n)=n^3$ 不是 $T(n)$ 的紧大O边界。现在可以选择 $g(n)=n^2$。我们已经看到，$T(n)$ 是 $O\big(g(n)\big)$，不过不能证明 $f(n)$ 是 $O\big(g(n)\big)$，因为 n^3 不是 $O(n^2)$。因此，n^3 不是 $T(n)$ 的紧大 O 边界。

3.5.4　简单性

在我们选择大O边界时，另一个目标就是函数表达式的简单性。与紧凑性不同，简单性有时候是种偏好问题。不过，一般还是可以按照如下标准认定函数 $f(n)$ 是简单的：

(1) 它只有一项；

(2) 这项的系数是1。

```
            int PowersOfTwo(int n)
            {
                int i;
(1)             i = 0;
(2)             while (n%2 == 0) {
(3)                 n = n/2;
(4)                 i++;
                }
(5)             return i;
            }
```

图3-5　计算一个正整数n中因数2的数量

✦ 示例 3.7

函数n^2是简单的，$2n^2$则不是简单的，因为系数不为1；而n^2+n也不是简单的，因为它包含了两项。

不过，也存在某些情况，其中大O紧凑性的上界和简单性的边界是相互冲突的目标。简单性边界并不能说明一切，以下就是一个例子，好在这样的情况在现实中很少出现。

✦ 示例 3.8

考虑一下图3-5中的`PowersOfTwo`函数，它会接受一个正参数n，并计量n被2整除的次数。也就是说，第(2)行的测试询问n是否为偶数，如果是，就在第(3)行的循环体中删除一个因数2。同样在这次循环中，我们递增`i`，而参数`i`的作用是计量我们从`n`原本的值中删除的因数2的个数。

设输入的大小就是n本身的值。`while`循环的循环体由两条语句组成，即第(3)行和第(4)行，因此可以说执行该循环体一次所需的时间为$O(1)$，也就是某个与n无关的不变时间量。如果该循环要执行m次，那么花在执行循环上的总时间就将是$O(m)$，或者是某个与m成比例的时间量。为单独执行第(1)行和第(5)行，以及进行第一次`while`循环条件的测试（从技术上讲不属于任何循环迭代的一部分），还要在这个量上加上$O(1)$或者某个常数。因此，该程序消耗的时间是$O(m)+O(1)$。根据低阶项可被忽略的规则，这一时间就是$O(m)$，除非$m=0$，此时这个时间就是$O(1)$。换个说法就是，在输入n上所花的时间与1加上2整除n的次数是成比例的。

在数学表达式中使用大O表示法

严格地讲，大O表达式在数学上正确的使用方式只有出现在“是”字后这一种情况，比如“$2n^2$是$O(n^3)$”。不过，在示例3.8以及本章余下的内容中，我们将直接把大O表达式当作加号以及其他算术运算符的操作数，比如表示为$O(n)+O(n^2)$。应将这样使用的大O表达式解释成“作为大O的某个函数”。例如$O(n)+O(n^2)$就表示“某个线性函数和某个二次函数的和”。此外，$O(n)+T(n)$应该解释为某个线性函数与某个特定函数$T(n)$的和。

---

n能被2整除多少次？对每个奇数n来说，答案为0。所以对每个奇数n，都有`PowersOfTwo`函数花的时间为$O(1)$。不过，当n是2的乘方，也就是说当n对某个k而言是2^k时，2能整除n的次数正好是k。当$n=2^k$时，可以在等式两边同时取以2为底的对数，得到$\log_2 n=k$。也就是说，m至多是n的对数，或者说$m=O(\log n)$。①

因此，可以说`PowersOfTwo`的运行时间是$O(\log n)$。这一边界满足了我们对简单性的定义。不过，还有更精确的方法来统计`PowersOfTwo`运行时间的上界，这就是说，它是函数$f(n)=m(n)+1$的大O，其中$m(n)$是n被2整除的次数。如图3-6所示，该函数一点都不简单。它的值在剧烈摆动，但从没有超过$1+\log_2 n$。

---

① 请注意，在大O表达式中说到对数时，是不需要指出底数的。原因在于，如果底数分别为a和b，那么$\log_a n=(\log_b n)(\log_a b)$。因为$\log_a b$是个常数，所以可以看到$\log_a n$和$\log_b n$只有一个常数因子的差别。因此，函数$\log_x n$对于任何不同底数$x$来说都互为大O，所以根据传递律，可以在大O表达式中用任意的$\log_b n$来代替$\log_a n$，其中b是不同于a的底数。

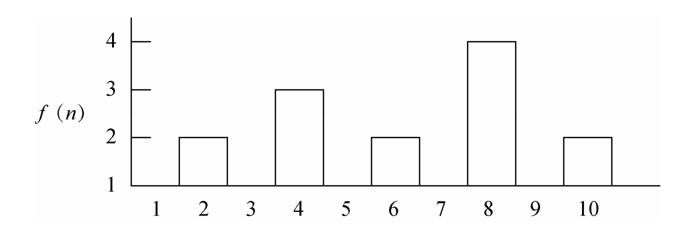


图3-6　函数 $f(n)=m(n)+1$，其中 $m(n)$ 是n被2整除的次数

因为`PowersOfTwo`的运行时间是$O\big(f(n)\big)$，而$\log n$又不是$O\big(f(n)\big)$，所以可以说$\log n$不是该程序运行时间的紧边界。另一方面，$f(n)$是紧边界，但它不简单。

运行时间中的对数

如果要考虑的算法需要处理积分（$\ln a=\int_1^a \frac{1}{x}\mathrm{d}x$），大家可能会因为它们出现在算法的分析中而感到惊讶。计算机科学家们通常会把“$\log n$”考虑为$\log_2 n$，而不是$\ln n$和$\lg n$。请注意，$\log_2 n$就是将n除以2直到得到1为止的次数，或者换句话说，是为了得到n，相乘的2的个数。大家可能很容易看出，$n=2^k$其实和说$\log_2 n=k$是一样的，只要在两边同时取以2为底的对数即可。

`PowersOfTwo`函数会尽可能多次地用2整除n，而且当n是2的乘方时，n能被2整除的次数就是$\log_2 n$。对数在对分治算法（就是在每个阶段将输入等分为两个部分，或者分为近似相等的两部分的算法，比如归并排序算法）的分析中会频繁地出现。如果我们一开始有大小为n的输入，那么将输入对半分，直到大小为1的阶段数是$\log_2 n$。或者，如果n不是2的乘方，就是比$\log_2 n$大的最小整数。

3.5.5　求和规则

假设某个程序由两部分组成，一部分耗费的时间是$O(n^2)$，而另一部分消耗的时间为$O(n^3)$。可以将这两个大O边界“相加”，从而得出整个程序的运行时间。在很多情况下（包括上述情况），通过应用如下求和规则，可以将大O表达式“相加”。

> 假设已知$T_1(n)$是$O\big(f_1(n)\big)$，而且$T_2(n)$是$O\big(f_2(n)\big)$。此外，假设f_2的增长率不大于f_1的增长率，也就是说，$f_2(n)$是$O\big(f_1(n)\big)$。那么就可以得出“$T_1(n)+T_2(n)$是$O\big(f_1(n)\big)^n$”的结论。

要证明这一规则，我们知道存在常数c_1、c_2、c_3、n_1、n_2和n_3，使得

(1) 如果$n\geqslant n_1$，则$T_1(n)\leqslant c_1f_1(n)$；

(2) 如果$n\geqslant n_2$，则$T_2(n)\leqslant c_2f_2(n)$；

(3) 如果$n\geqslant n_3$，则$f_2(n)\leqslant c_3f_1(n)$。

设n_0是n_1、n_2和n_3中最大的那个，则当$n\geqslant n_0$时，(1)、(2)和(3)都成立。因此，对$n\geqslant n_0$，有

$$T_1(n)+T_2(n)\leqslant c_1f_1(n)+c_2f_2(n)$$

如果使用(3)提供$f_2(n)$的上边界，那么完全可以消去$f_2(n)$，并得出

$$T_1(n)+T_2(n) \leqslant c_1 f_1(n) + c_2 c_3 f_1(n)$$

因此，如果定义c为$c_1+c_2c_3$，就证明了对于所有的$n \geqslant n_0$，有

$$T_1(n)+T_2(n) \leqslant cf_1(n)$$

这一命题刚好就是我们需要得出的结论——$T_1(n)+T_2(n)$是$O\big(f_1(n)\big)$。

✦ 示例 3.9

考虑一下图3-7中的程序段。该程序会使A成为n阶单位矩阵。第(2)行至第(4)行在该$n \times n$二维数组的每个单元中都放上0，接着第(5)行和第(6)行会在从`A[0][0]`到`A[n-1][n-1]`的对角线线上的位置中放入1。结果就形成了具有对于任意$n \times n$矩阵M都有如下属性的单位矩阵A。

```
              A × M = M × A = M
(1)         scanf("%d",&n);
(2)         for (i = 0; i < n; i++)
(3)             for (j = 0; j < n; j++)
(4)                 A[i][j] = 0;
(5)         for (i = 0; i < n; i++)
(6)             A[i][i] = 1;
```

图3-7 创建单位矩阵A的程序段

第(1)行会读取n，花的时间为$O(1)$，也就是某个和n值无关的固定时间量。第(6)行中的赋值语句花的时间也是为$O(1)$，第(5)行和第(6)行的循环要进行n次，在该循环上花的总时间就是$O(n)$。类似地，第(4)行中的赋值语句花的时间是$O(1)$。第(3)行和第(4)行的循环要进行n次，花费的总时间为$O(n)$。第(2)行至第(4)行的外层循环要执行n次，在每次迭代中花费的时间为$O(n)$，所以总时间就是$O(n^2)$。

因此，图3-7所示程序的运行时间就是$O(1)+O(n^2)+O(n)$，分别表示语句(1)、第(2)行至第(4)行的循环，以及第(5)行和第(6)行的循环。更正式地讲，如果以下几点同时成立：

$T_1(n)$是第(1)行所花的时间；

$T_2(n)$是第(2)行至第(4)行所花的时间；

$T_3(n)$是第(5)行和第(6)行所花的时间。

那么可以得出如下结论。

$T_1(n)$是$O(1)$；

$T_2(n)$是$O(n^2)$；

$T_3(n)$是$O(n)$。

因此我们需要$T_1(n)+T_2(n)+T_3(n)$的上界，从而得出整个程序的运行时间。

因为常数1显然是$O(n^2)$，所以可以应用求和规则得出$T_1(n)+T_2(n)$是$O(n^2)$。因为n是$O(n^2)$，就可以对$\big(T_1(n)+T_2(n)\big)$和$T_3(n)$应用求和规则，从而得出$T_1(n)+T_2(n)+T_3(n)$是$O(n^2)$。也就是说，图3-7所示的整个程序段的运行时间是$O(n^2)$。通俗地讲，就是整个程序几乎将所有的运行时间都花在了第(2)行至第(4)行的循环上，正如我们从以下事实中很容易就能想到的：对于很大的n，矩阵的面积n^2要比由n个单元组成的对角线大得多。

示例3.9应用了“低阶项不产生影响”这条规则，因为我们舍弃了1和n这两项比n^2次数更低的多项式。不过，求和规则不仅仅能让我们舍弃低阶项。如果有任意多个相同的大O常数项，

比如有一列10个赋值语句，每个赋值语句所花的时间都是$O(1)$，那么就可以将这10个$O(1)$“加起来”，得到$O(1)$。不那么严格地讲就是，10个常数的和还是个常数。要知道原因，请注意1是$O(1)$，所以10个$O(1)$中任何一个都可以“被加到”其他任意一个$O(1)$上，从而得出$O(1)$这个结果。我们可以不断合并项，直到只剩下$O(1)$为止。

不过，必须要小心，不要把某个像$O(1)$这样的“常数”项，与这些随输入大小变化的项弄混了。例如，我们有可能错误地认为，每进行一次图3-7中第(5)行和第(6)行所示的循环，花的时间为$O(1)$，而该循环总共循环了n次，所以第(5)行和第(6)行的总运行时间就是$O(1)+O(1)+O(1)+\cdots+\left(n个O(1)\right)$，而求和规则告诉我们两个$O(1)$的和也是$O(1)$，这样，根据归纳法就可以得出结论：任意多个$O(1)$的和都是$O(1)$。但是，在这个程序中，$n$不是常数，它会因输入大小而异。因此，我们没法通过多次应用求和规则推断出n个$O(1)$具有任何特殊的值。当然，如果真要考虑这个问题，那么我们知道n个c的和（其中c是某个常数）是cn，该函数的大O形式是$O(n)$，而这就是第(5)行和第(6)行真正的运行时间。

3.5.6　不相称函数

任意两个函数$f(n)$和$g(n)$可由大O相比较。也就是说，要么$f(n)$是$O\left(g(n)\right)$，要么$g(n)$是$O\left(f(n)\right)$。或者二者互为对方的大O，因为我们看到过，$2n^2$和n^2+3n这两个函数就是这种互为大O的关系。这种情况是很不错的。不过不巧的是，也有一些不相称的函数对，它们之间不存在任何大O关系。

✦ 示例 3.10

考虑如下函数

$$f(n)=\begin{cases} n, & n\text{ 为奇数} \\ n^2, & n\text{ 为偶数} \end{cases}$$

也就是，$f(1)=1$，$f(2)=4$，$f(3)=3$，$f(4)=16$，$f(5)=5$，等等。类似地，假设有函数

$$g(n)=\begin{cases} n^2, & \text{为奇数} \\ n, & \text{为偶数} \end{cases}$$

那么$f(n)$不可能是$O\left(g(n)\right)$，因为那些偶数n。因为如我们在3.4节中看到的，n^2绝对不是$O(n)$。类似地，$g(n)$也不可能是$O\left(f(n)\right)$，因为那些奇数n，当n为奇数时，g的值比f的值增长得更快。

3.5.7　习题

(1) 证明如下命题。

(a) 如果$a\leqslant b$，那么n^a是$O(n^b)$。

(b) 如果$a>b$，那么n^a不是$O(n^b)$。

(c) 如果$1<a\leqslant b$，那么a^n是$O(b^n)$。

(d) 如果$1<b<a$，那么a^n不是$O(b^n)$。

(e) 对任意a和任意$b>1$，n^a是$O(b^n)$。

(f) 对任意b和任意$a>1$，a^n是$O(n^b)$。

(g) 对任意a和任意$b>0$，$(\log n)^a$是$O(n^b)$。

(h) 对任意b和任意$a>0$，n^a不是$O\left((\log n)^b\right)$。

(2) 证明：$f(n)+g(n)$ 是 $O\big(\max\big(f(n),g(n)\big)\big)$ 。

(3) 假设 $T(n)$ 是 $O\big(f(n)\big)$ ，且 $g(n)$ 是某个值不为负的函数。证明：$g(n)T(n)$ 是 $O\big(g(n)f(n)\big)$ 。

(4) 假设 $S(n)$ 是 $O\big(f(n)\big)$ ，且 $f(n)$ 是 $O\big(g(n)\big)$ ，而且这些函数对任意n都不为负值。证明：$S(n)T(n)$ 是 $O\big(f(n)g(n)\big)$ 。

(5) 假设 $f(n)$ 是 $O\big(g(n)\big)$ 。证明：$\max\big(f(n),g(n)\big)$ 是 $O\big(g(n)\big)$ 。

(6) * 证明：如果 $f_1(n)$ 和 $f_2(n)$ 都是某个函数 $T(n)$ 的紧边界，那么 $f_1(n)$ 和 $f_2(n)$ 互为对方的大O。

(7) * 证明：对于图3-6所示的函数 $f(n)$ ，$\log_2 n$ 不是 $O\big(f(n)\big)$ 。

(8) 在图3-7所示的程序中，通过先在矩阵中每个位置放上0，然后在对角线上放上1，我们创建了一个单位矩阵。将第(4)行的测试改为询问是否有 $i = j$ ，如果是，则在`A[i][j]`中放上1，如果不是，则放上0，这样修改后似乎能更快地完成这一工作。然后我们还可以删除第(5)行和第(6)行。

　(a) 写出这一程序。

　(b) * 考虑图3-7中的程序以及自己为问题(a)编写的程序。作出示例3.1中那样的简化假设，计算两个程序分别耗费了多少个时间单位。哪个程序更快？用不同大小的二维数组运行这两个程序，并绘制它们的运行时间曲线。

3.6　分析程序的运行时间

掌握了大O的概念，以及3.4节和3.5节中介绍的那些处理大O表达式的规则之后，我们将要学习如何获得常见程序运行时间的大O上界。只要有可能，我们将只考虑那些不含函数调用（除了诸如`printf`那样的库函数）的程序，将含有函数调用的问题留待3.8节及以后的内容中介绍。

我们不指望能够分析任意程序，因为有关运行时间的问题可能是非常难的数学问题。另一方面，只要了解一些简单的规则，我们就能够计算出实践中遇到的多数程序的运行时间。

3.6.1　简单语句的运行时间

这里要求读者接受这样一个原则，即某些对数据的简单操作可以在 $O(1)$ 时间内完成，也就是说，这个时间是和输入大小无关的。C语言中的这些基本操作包括：

(1) 算术运算（比如`+`或`%`）；

(2) 逻辑运算（比如`&&`）；

(3) 比较运算（比如`<=`）；

(4) 结构体存取操作（比如`A[i]`这样的数组索引，或者跟在指针后的`->`运算符）；

(5) 简单的赋值（比如将某个值复制到某个变量中）；

(6) 对库函数（比如`scanf`、`printf`）的调用。

对这一原则的验证需要对常见计算机的机器指令（初始步骤）进行详细研究。我们很容易看出，之前描述的每种操作都只需要少量机器指令便可完成，通常只需要1条或2条指令。

因此，在C语言中有好几种语句都能在 $O(1)$ 时间内执行完，也就是说，可以在与输入无关的某个时间段内执行完。这些简单语句包括：

(1) 表达式中不涉及函数调用的赋值语句；

(2) 读语句；

(3) 不需要调用函数确定参数值的写语句；

(4) 跳转语句`break`、`continue`、`goto`和`return`表达式，其中表达式不含函数调用。

在第(1)到第(3)条中，这些语句都是由有限数量的基本操作构成的，每个操作花的时间都是$O(1)$。由求和规则可知，整个语句花的时间是$O(1)$。当然，语句对应的时间常数要比单个操作对应的常数大，不过我们已经知道，无论如何也不能将具体的常数与C语言语句的运行时间关联起来。

✦ 示例 3.11

我们在示例3.9中看到，图3-7中第(1)行的读语句，以及第(4)行和第(6)行中的赋值，每一行花费的时间都是$O(1)$。再看一个例子，即图3-8中展示的选择排序程序段。第(2)、(5)、(6)、(7)和第(8)行，每一行花费的时间都是$O(1)$。

```
(1)         for (i = 0; i < n-1; i++) {
(2)             small = i;
(3)             for (j = i+1; j < n; j++)
(4)                 if (A[j] < A[small])
(5)                     small = j;
(6)             temp = A[small];
(7)             A[small] = A[i];
(8)             A[i] = temp;
            }
```

图3-8　选择排序程序段

我们经常会看到由连续执行的简单语句构成的程序块。如果每条语句的运行时间都是$O(1)$，那么根据求和规则，整个程序块花费的时间也是$O(1)$。也就是说，任意固定多个$O(1)$的和还是$O(1)$。

✦ 示例 3.12

图3-8中的第(6)行到第(8)行形成了一个程序块，因为它们永远是连续执行的。由于每一行花的时间都是$O(1)$，所以第(6)行到第(8)行的程序块所花的时间也是$O(1)$。

请注意，不应该把第(5)行算在程序块中，因为它是第(4)行`if`语句的一部分。也就是说，有时候即便不执行第(5)行，第(6)行至第(8)行也会执行。

3.6.2　简单for循环的运行时间

在C语言中，很多`for`循环的构成包括初始化指标变量为某个值的语句，以及每进行一次循环就将该标量递增1的语句。当该指标达到某个限制后，`for`循环就终止了。例如，图3-8中第(1)行的`for`循环使用了指标变量`i`。每进行一次循环，它就将`i`递增1，而当`i`达到$n-1$时，迭代就停止了。

在C语言中，还有更复杂的`for`循环，其行为更类似`while`语句，这些循环迭代的次数是不可预知的。本节后面将会介绍这种循环。不过在这里，还是将注意力集中在形式简单的`for`循环上，在这种`for`循环中，最终值和初始值之间的差，除以指标变量每次递增的量，就可以得出循环了多少次。这种计数是精确的，除非还存在一些通过跳转语句退出循环的方式，否则这在任何情况下都是迭代次数的上界。例如，图3-8中`for`循环的第1行会迭代$\left((n-1)-0\right)/1=n-1$次，因为0是$i$的初始值，$n-1$是$i$达到的最高值（即当$i$达到$n-1$时，循环就会终止，$i=n-1$时不会发生迭代），而且循环每次迭代`i`都会增加1。

要为for循环的运行时间找出边界，必须先找到循环体进行一次迭代所花时间的上界。请注意，进行一次迭代的时间包括递增循环指标（比如图3-8第(1)行中的递增语句i++）所花的时间$O(1)$，以及比较循环指标与上限（比如图3-8第(1)行中的测试语句i<n-1）所花的时间$O(1)$。除了循环体为空的异常情况，其他所有情况下的这些$O(1)$都可以根据求和规则舍弃掉。

在最简单的情况，也就是循环体每次迭代所花的时间均相同的情况下，可以用循环体的大O上界乘上循环的次数。严格地说，还必须加上初始化循环指标的时间$O(1)$，以及第一次比较循环指标和上限的时间$O(1)$。不过，除非有可能不执行循环，否则初始化循环和测试上限的时间都是根据求和规则可被舍弃的低阶项。

✦ 示例 3.13

考虑图3-7第(3)行和第(4)行中的for循环，也就是

```
(3)         for (j = 0; j < n; j++)
(4)             A[i][j] = 0;
```

我们知道第(4)行花的时间为$O(1)$。显然，我们要进行n次循环，这可以由第(3)行找到的上限减去下限再加上1来确定。因为循环体，也就是第(4)行，花费的时间为$O(1)$，所以可以忽略递增j的时间$O(1)$以及比较j与n的时间$O(1)$。因此，第(3)行和第(4)行的运行时间为n与$O(1)$的积，也就是$O(n)$。

类似地，可以确定由第(2)行至第(4)行构成的外层循环的运行时间边界，外层循环如下。

```
(2)         for (i = 0; i < n; i++)
(3)             for (j = 0; j < n; j++)
(4)                 A[i][j] = 0;
```

我们已经得到第(3)行和第(4)行的循环所花的时间为$O(n)$。因此，可以忽略递增i的时间$O(1)$以及每次迭代时测试是否有$i<n$所花的时间$O(1)$，并得出外层循环每次迭代所花的时间为$O(n)$。外层循环初始化i=0，以及第$(n+1)$次$i<n$的条件测试花的时间都是$O(1)$，而且都可以忽略。最终，我们看到外层循环要循环n次，而每次迭代的时间都是$O(n)$，因此总运行时间就是$O(n^2)$。

✦ 示例 3.14

现在来考虑图3-8第(3)行到第(5)行中的for循环。在这里，循环体是if语句，是我们接下来将要了解如何进行分析的结构。不难推断出第(4)行花费时间$O(1)$执行测试，第(5)行如果执行的话也会花费时间$O(1)$，因为它是不含函数调用的赋值语句。因此，不管第(5)行是否执行，执行for循环循环体所花的时间都为$O(1)$，循环中的递增和测试增加的时间都是$O(1)$，所以循环进行一次迭代的总时间也只是$O(1)$。

现在我们必须计算进行循环的次数。迭代次数是与输入大小n无关的。而公式“最后的值减去初始值除以递增量”告诉我们，$\big(n-(i+1)\big)/1$，或者说$n-i-1$，是循环迭代的次数。严格地说，该公式只有在$i<n$时才成立。好在我们从图3-8的第(1)行可以看出，除非$i\leqslant n-2$，否则我们不会进入第(2)至第(8)行的循环体。因此，我们不仅知道了$n-i-1$是循环迭代的次数，而且知道了这个数值不可能为0。由此可以得出该循环所花的时间为$(n-i-1)\times O(1)$，或者说是$O(n-i-1)$。[①]此处不必加上初始化j所花的时间$O(1)$，因为已知$n-i-1$不可能为0。如果看不

① 从技术上讲，我们没有讨论过应用到多变量函数上的大O运算符。在这种情况中，可以将$O(n-i-1)$说成是“最多为某个常数乘以$n-i-1$”。也就是说，可以将$n-i-1$视为某个单变量函数的替代物。

出 $n-i-1$ 为正的话，就必须将运行时间的上界写为 $O\big(\max(1, n-i-1)\big)$ 。

3.6.3　选择语句的运行时间

if-else选择语句具有如下形式：

```
if (<condition>)
    <if-part>
else
    <else-part>
```

其中

(1) 条件是待评估的表达式；

(2) if部分的语句只有在条件为真（表达式的值不为0）时才执行；

(3) else部分的语句只有在条件为假（评估为0）时才执行，else后的<else-part>是可选的。

只要条件中没有函数调用，不管条件多么复杂，都只需要计算机执行一定量的基本操作。因此，条件评估所花的时间为 $O(1)$ 。

假设在条件中没有函数调用，而且if部分和else部分分别具有大O上界 $f(n)$ 和 $g(n)$ 。还假设 $f(n)$ 和 $g(n)$ 不会都为0，也就是说，尽管else部分可能不存在，但if部分是不会为空的。我们将确定两部分都为空的时候会发生什么留作本节的习题。

如果 $f(n)$ 是 $O\big(g(n)\big)$ ，那么可以将 $O\big(g(n)\big)$ 作为选择语句运行时间的上界。原因包括：

(1) 可以忽略条件所花的时间 $O(1)$ ；

(2) 如果else部分执行，就可知 $g(n)$ 是运行时间的边界；

(3) 如果if部分（而不是else部分）执行，那么运行时间将是 $O\big(g(n)\big)$ ，因为 $f(n)$ 是 $O\big(g(n)\big)$ 。

类似地，如果 $g(n)$ 是 $O\big(f(n)\big)$ ，就可以通过 $O\big(f(n)\big)$ 确定选择语句运行时间的边界。请注意，当else部分不存在时（情况也常常是这样）， $g(n)$ 为0，就肯定是 $O\big(f(n)\big)$ 。

当f和g之间不存在大O关系时，问题出现了。我们知道if部分或else部分肯定有一种要执行，但不可能都执行，所以运行时间的安全上界就是 $f(n)$ 和 $g(n)$ 中的较大者。正如我们在示例3.10中看到的，二者谁比较大可能取决于n。因此，要将选择语句的运行时间表示为 $O\Big(\max\big(f(n), g(n)\big)\Big)$ 。

✦　示例 3.15

正如我们在示例3.12中看到的，图3-8中第(4)行和第(5)行是选择语句，其中第(5)行是if部分，所花时间为 $O(1)$ ，而不存在else部分（也就是所花时间为0）。因此， $f(n)$ 是1且 $g(n)$ 是0。由于 $g(n)$ 是 $O\big(f(n)\big)$ ，可以得出 $O(1)$ 是第(4)行和第(5)行运行时间的上界。请注意，在第(4)行执行测试`A[j]<A[small]`的时间 $O(1)$ 可以忽略。

✦　示例 3.16

图3-9所示的代码段是个更为复杂的例子，它执行的（相对无意义的）任务是将矩阵A置为0，或是将矩阵的对角线置为1。一如我们在示例3.13中所了解的，第(2)行至第(4)行的运行时间是

$O(n^2)$，而第(5)行和第(6)行的运行时间是$O(n)$。因此这里的$f(n)$是n^2，$g(n)$是n。因为n是$O(n^2)$所以可以忽略else部分的时间，并将$O(n^2)$作为图3-9中整个程序段运行时间的边界。也就是说，我们不知道第(1)行的条件是否将为真或者什么时候将为真，不过唯一安全的上界是从最坏的假设中得出的，即条件为真而且if部分执行了。

```
(1)        if (A[1][1] == 0)
(2)            for (i = 0; i < n; i++)
(3)                for (j = 0; j < n; j++)
(4)                    A[i][j] = 0;
           else
(5)            for (i = 0; i < n; i++)
(6)                A[i][i] = 1;
```

图3-9 if-else选择语句的示例

3.6.4 程序块的运行时间

前文已经提到，一系列赋值、读、写操作，每一次操作的时间都是$O(1)$，总时间也是$O(1)$。一般的情况是，必须能将一系列语句（其中有一些是*复合语句*，也就是选择语句或循环）组合起来。这样一系列简单的复合语句就是*程序块*（block）。要计算程序块的运行时间，需要对程序块中每条（可能是复合的）语句的大O上界求和。好在可以使用求和规则消除和中的一些项。

✦ 示例 3.17

在图3-8的选择排序程序段中，可以将外层循环的循环体（也就是第(2)行至第(8)行）视为一个程序块。该程序块由5条语句组成。

(1) 第(2)行的赋值语句。

(2) 第(3)行、第(4)行和第(5)行的循环。

(3) 第(6)行的赋值语句。

(4) 第(7)行的赋值语句。

(5) 第(8)行的赋值语句。

请注意，第(4)和第(5)行的选择语句以及第(5)行的赋值在程序块这一级是不可见的，它们已经隐藏在更大的语句，也就是第(3)行至第(5)行的循环中了。

我们知道，4条赋值语句每条所花的时间都是$O(1)$。在示例3.14中，已经了解到该程序块中第2条语句（也就是第(3)行至第(5)行）的运行时间是$O(n-i-1)$。因此，该程序块的运行时间是：

$$O(1)+O(n-i-1)+O(1)+O(1)+O(1)$$

因为1是$O(n-i-1)$（回想一下，我们还推导出i从不会大于$n-2$），所以可以通过求和规则消除所有的$O(1)$项。因此，整个程序块的运行时间就是$O(n-i-1)$。

再看一个例子，考虑一下图3-7中的程序段。它可被视为由3条语句组成的单一程序块。

(1) 第(1)行的读语句。

(2) 第(2)行至第(4)行的循环。

(3) 第(5)行和第(6)行的循环。

我们知道，第(1)行花的时间为$O(1)$。从示例3.13可知，第(2)行至第(4)行花的时间是$O(n^2)$，第(5)行和第(6)行花的时间是$O(n)$。所以整个程序块的运行时间就是：

$$O(1)+O(n^2)+O(n)$$

根据求和规则，由 $O(n^2)$ 可以消去 $O(1)$ 和 $O(n)$ 。因此可以得出图3-7中程序段的运行时间为 $O(n^2)$ 。

3.6.5　复杂循环的运行时间

在C语言中，有一些while循环、do-while循环和for循环并未提供显式的计数变量。对于这些循环，一部分分析工作就是要找到为循环迭代次数提供上界的参数。这些证明过程通常都遵循我们在2.5节中了解的模式。也就是说，通过对循环次数的归纳证明某个命题，而该命题表明在迭代次数达到某个限制后，循环条件一定会变为假。

我们还必须建立执行一次循环迭代所花时间的边界。因此，可以对循环体加以研究，并获得其执行的边界。为了实现这个目标，必须在循环体执行后加上测试条件的时间 $O(1)$ ，不过除非循环体不存在，否则我们都会忽略该 $O(1)$ 项。通过用迭代次数的上界乘以一次迭代所花时间的上界，可以得到循环运行时间的边界。从技术上讲，如果该循环是for循环或while循环，而不是do-while循环，就必须将进入循环体之前第一次测试条件所需的时间包含在内。不过，这个 $O(1)$ 经常是可以忽略掉的。

✦　示例 3.18

考虑如图3-10所示的程序段。该程序会搜索数组A[0..n-1]，找出该数组中的元素x。

```
(1)        i = 0;
(2)        while(x != A[i])
(3)            i++;
```

图3-10　线性查找的程序段

图3-10中第(1)行和第(3)行的两条赋值语句的运行时间均为 $O(1)$ 。第(2)和第(3)行的while循环可能会执行n次，但不会超过n次，因为我们假设x确实是数组元素之一。因为第(3)行的循环体所需时间为 $O(1)$ ，所以该while循环的运行时间就是 $O(n)$ 。根据求和规则，整个程序段的运行时间为 $O(n)$ ，因为这是第(1)行的赋值语句以及整个while循环所花的最大时间。在第6章中，我们还将看到这种 $O(n)$ 程序是如何被使用二叉查找的 $O(\log n)$ 程序所代替的。

3.6.6　习题

(1) 对开头为for (i = a; i <= b; i++)的for循环，用a和b的函数表示其循环次数。对开头为for (i = a; i <= b; i--)的for循环又是怎样表示的呢？对开头为for(i = a; i <= b; i = i+c)的for循环呢？

(2) 给出某个普通的选择语句if(C) {}运行时间的大O上界，其中C是不涉及任何函数调用的条件。

(3) 给出某个普通的while循环while(C) {}运行时间的大O上界，其中C是不涉及任何函数调用的条件。

(4) * 给出C语言switch语句运行时间的规则。

(5) 给出我们能确定哪条分支被执行的选择语句运行时间的规则，比如

```
if (1==2)
    something O(f(n));
else
    something O(g(n));
```

(6) 给出循环开始前条件已知为假的退化`while`循环（degenerate while-loop）运行时间的规则，比如

```
while (1 != 1)
    something O(f(n));
```

3.7　边界运行时间的递归规则

在3.6节中，我们简略地描述了一些规则，它们用程序结构各部分的运行时间来定义整个程序结构的运行时间。例如，我们说过`for`循环的运行时间大致等于循环体所花的时间乘以迭代的次数。隐藏在这些规则背后的概念是，程序是使用归纳规则构成的，复合语句（循环、选择和其他由子语句组成的语句）通过这些规则由诸如赋值、读、写和跳转语句这样的简单语句组成。这些归纳规则涵盖循环的形成、选择语句及程序块等一系列复合语句。

我们要将一些构建C语言语句的句法规则表述为递归定义。这些规则符合经常出现在C语言教材中的那些定义C语言的语法规则。我们在第11章中还将看到，语法可以用作简洁递归表示法，来指明编程语言句法（syntax）。

更具防御性的程序设计

如果大家只是因为相信示例3.18中的数组A总会存在元素x，就认为它总会存在，那就太天真了。请注意，如果数组中不存在x，图3-10中的循环将最终会出错，因为它要试着访问一个超过数组上限的数组元素。

好在有一种简单的方法可以避免这一错误，而且不会给循环的每次迭代增加很多时间。我们允许数组末尾有第$n+1$个单元，而在开始循环前，将x放在该单元中。那么确实能确定x会出现在数组中的某个位置。当循环结束后，我们会测试是否有$i=n$。如果是，那么x并非真正在数组中，我们会穿过数组到达作为哨兵（sentinel）的x的副本。如果$i<n$，那么i就表示x出现的位置。带有这种保护功能的程序如下所示。

```
A[n] = x;
i = 0;
while (x != A[i])
    i++;
if (i == n) /* do something appropriate to the case
                      that x is not in the array */
else /* do something appropriate to the case
                      that x is found at position i */
```

依据。

C语言中的简单语句如下。

(1) 表达式。包括赋值语句以及读和写语句，后者是对`printf`和`scanf`等函数的调用；

(2) 跳转语句。包含`goto`、`break`、`continue`和`return`；

(3) 空语句。

请注意，在C语言中，简单语句都是以分号结尾的，我们要将分号视为这些语句的一部分。

归纳。

以下规则让我们可以用较小的语句来构建语句。

(1) `while`语句。如果S是语句，而C是条件（带有算术值的表达式），那么

while (C) S

是语句。只要C为真（具有非0的值），循环体S就会执行。

(2) do-while语句。如果S是语句，而C是条件，那么

do S **while**（C）

是语句。do-while循环和while循环类似，只不过do-while循环的循环体S至少会执行一次。

(3) for语句。如果S是语句，而E_1、E_2和E_3是表达式，那么

for（E_1；E_2；E_3）S

是语句。第一个表达式E_1会进行一次评估，并指定循环体S的初始化。第二个表达式E_2是对循环终止的测试，会在每次迭代前进行评估。如果它的值不为0，那么循环体就会执行，否则该for循环就将终止。第三个表达式E_3会在每次迭代后进行评估，并为循环的下一次迭代指定重初始化（递增）。例如，如下常见的for循环

for (i = 0; i < n; i++) S

其中S会迭代n次，对应i的值分别为1、2、3、…、$n-1$。在这里，i = 0是初始化，i < n是终止测试，i++是重初始化。

(4) 选择语句。如果S_1和S_2是语句，而C是条件，那么

if（C）S_1 **else** S_2

是语句，而且

if（C）S_1

也是语句。在第一种情况中，如果C为真（非0），就执行S_1，否则就执行S_2。在第二种情况中，只有当C为真，才执行S_1。

(5) 程序块。如果S_1、S_1、…、S_n都是语句，那么

$$\{S_1 \ S_2 \cdots S_n\}$$

也是语句。

我们在上面没有列出开关语句，它形式复杂，但在分析运行时间时可以被当作嵌套的选择语句。

利用上述对语句的递归定义，就可以通过分辨程序的组成部分来解析程序。也就是说，首先有简单的语句，再进一步将这些简单的语句组成更大的复合语句。

✦ 示例 3.19

考虑图3-11所示的选择排序程序段。作为根据，第(2)行、(5)行、(6)行、(7)行和第(8)行的每次赋值都各为一条语句；而第(4)行和第(5)行组成了选择语句；第(3)行至第(5)行又组成了for语句；然后第(2)行至第(8)行组成了一个程序块；最后，整个程序段也是for语句。

```
(1)        for (i = 0; i < n-1; i++) {
(2)            small = i;
(3)            for (j = i+1; j < n; j++)
(4)                if (A[j] < A[small])
(5)                    small = j;
(6)            temp = A[small];
(7)            A[small] = A[i];
(8)            A[i] = temp;
           }
```

图3-11　选择排序程序段

3.7.1 程序结构的树表示

我们可以用如图3-12所示的树表示程序的结构。树叶（那些圆圈）是简单语句，而其他的节点则表示复合语句。[①]节点会被标记上它们所表示结构的种类，以及构成该节点所表示简单语句或复合语句的代码行。从每个表示复合语句的节点*N*都会向下引出到达其“子节点”的连线。节点*N*的子节点表示构成*N*所表示复合语句的那些子语句。这样的树就称为程序的结构树。

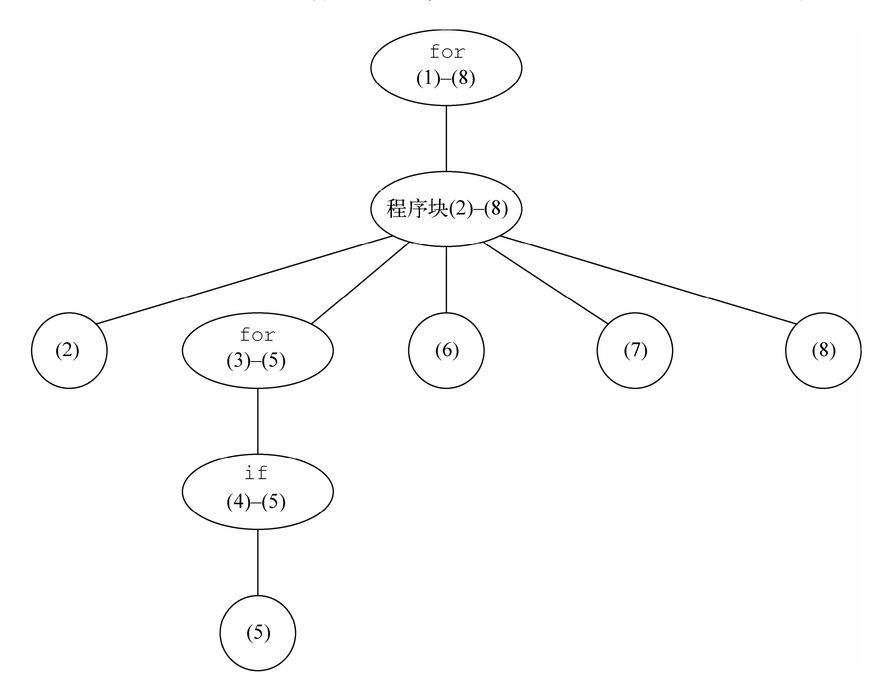


图3-12　表示语句组合的树

✦ 示例 3.20

图3-12是图3-11所示程序的结构树。每个圆圈分别是表示图3-11中5条赋值语句的树叶。我们在图3-12中没有说明这5条语句是赋值语句。

在树的顶端（也就是“根”）是表示第(1)至第(8)行整个程序段的节点。`for`循环的循环体是由第(2)行至第(8)行组成的程序块。[②]该程序块是用根节点下方的节点表示的。而这个表示程序块的节点又有5个子节点，分别表示该程序块的5条语句。其中第(2)、(6)、(7)和第(8)行这4条是赋值语句，而第5条是第(3)行至第(5)行的`for`循环。

第(3)行至第(5)行表示`for`循环的节点又有表示其循环体（就是第(4)行和第(5)行的`if`语句）的子节点。而表示第(4)行和第(5)行`if`语句的节点又具有表示其组成语句（第(5)行的赋值语句）的子节点。

3.7.2 攀爬结构树以确定运行时间

正如递归构建的程序结构那样，我们可以使用类似的递归方法来定义程序运行时间的大O

① 我们将在第5章中详细讨论树。

② 更为详细的结构树还有表示`for`循环初始化表达式、终止测试表达式和重初始化表达式的子节点。

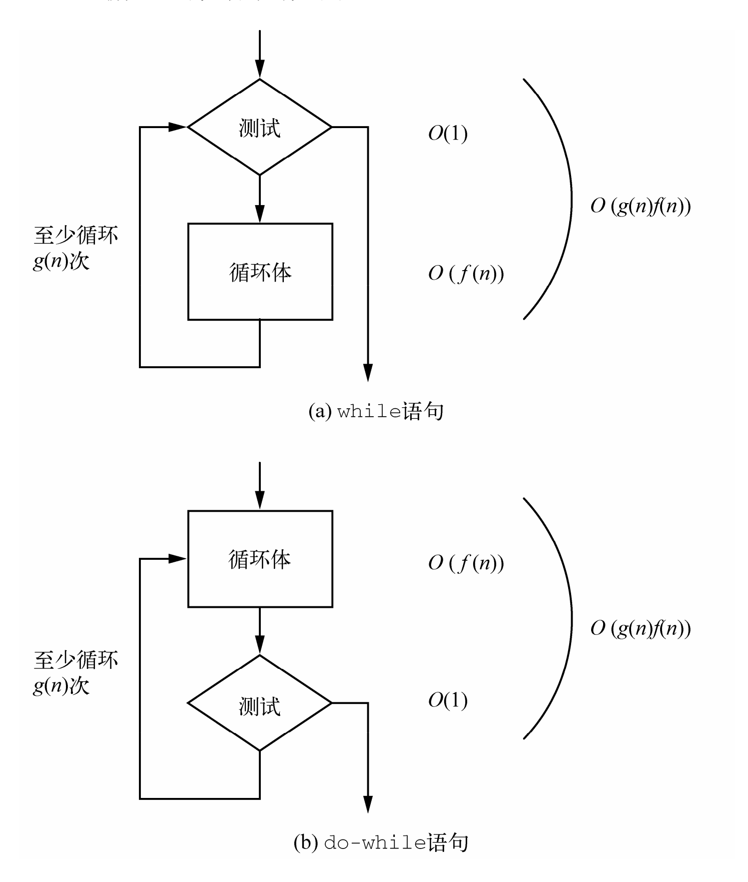
上界。就像在3.6节中那样，我们假定在下列几类表达式中都不存在函数调用。(1) 构成赋值语句、打印语句、选择语句条件的表达式；(2) 构成while循环、for循环和do-while循环条件的表达式；(3) for循环初始化或重初始化的表达式。唯一的例外是对诸如printf这样的读函数或写函数的调用。

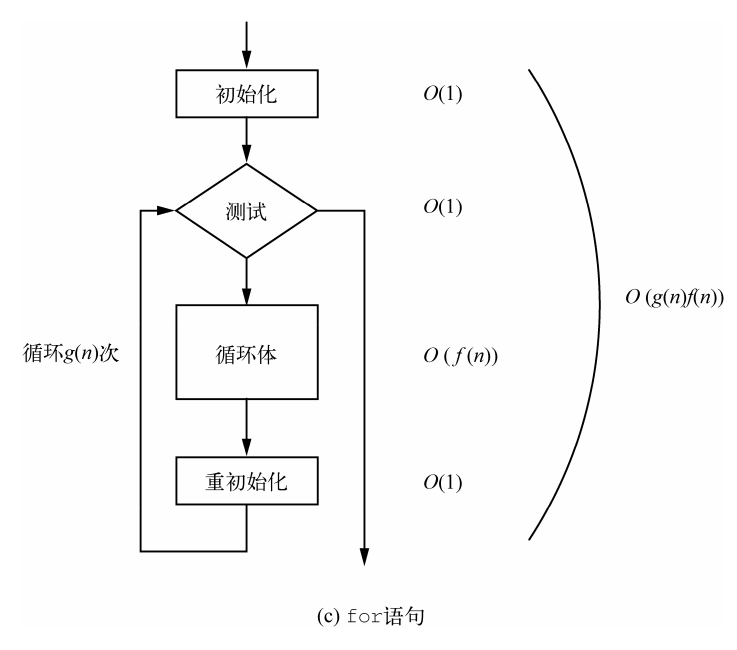
依据。简单语句（也就是赋值、读、写或跳转语句）的边界是 $O(1)$ 。

归纳。对于我们已经讨论过的5种复合结构，计算其运行时间的规则如下。

(1) while语句。设 $O\big(f(n)\big)$ 是while语句循环体的运行时间上界， $f(n)$ 是通过递归地应用这些规则得到的。再假设 $g(n)$ 是循环次数的上界。那么 $O\Big(1+\big(f(n)+1\big)g(n)\Big)$ 就是整个while循环的运行时间上界，其中 $O\big(f(n)+1\big)$ 是循环体加上循环体后测试的运行时间上界。开头那个多出来的1表示循环开始前的第一次测试。在 $f(n)$ 和 $g(n)$ 都至少为1（或者如果不定义其值为1，则其值为0，我们就可以定义它们为1）的平常情况下，可以将该while循环的运行时间记为 $O\big(f(n)g(n)\big)$ 。这一运行时间的通用公式如图3-13a所示。

(2) do-while语句。如果 $O\big(f(n)\big)$ 是循环体运行时间的上界，且 $g(n)$ 是循环次数的上界，那么 $O\Big(\big(f(n)+1\big)g(n)\Big)$ 就是该do-while循环的运行时间上界。这里“+1”表示的是循环每次迭代之末计算和测试循环条件的时间。请注意，对do-while循环来说， $g(n)$ 总是至少为1。在对所有n都有 $f(n)\geqslant 1$ 的情况中，do-while循环的运行时间为 $O\big(f(n)g(n)\big)$ 。图3-13b表示了计算普通情况下的do-while循环运行时间的方法。





(c) for语句

图3-13 计算不含函数调用的循环语句的运行时间

(3) for语句。如果$O\big(f(n)\big)$是循环体运行时间的上界，且g(n)是循环次数的上界，那么for语句运行时间的上界就是$O\big(\big(1+f(n)+1\big)g(n)\big)$。因子$f(n)+1$表示每进行一次循环所花的时间。开头的“1+”表示第一次初始化，以及第一次测试为负从而导致循环体不执行这种可能。在$f(n)$和$g(n)$都至少为1，或者可重新定义为至少是1的一般情况下，for语句的运行时间是$O\big(f(n)g(n)\big)$，如图3-13c所示。

(4) 选择语句。如果$O\big(f_1(n)\big)$和$O\big(f_2(n)\big)$分别是if部分和else部分的运行时间（如果没有else部分，则$O\big(f_2(n)\big)$为0），那么选择语句运行时间的上界就是$O\big(1+\max\big(f_1(n),f_2(n)\big)\big)$。“1+”表示条件测试，在$f_1(n)$和$f_2(n)$至少有一个为正数的一般情况下，这个“1+”是可以忽略的。此外，如果$f_1(n)$和$f_2(n)$中有一个是另一个的大O，那么该表达式如3.5节习题(5)中所述那样可以简化为二者中的较大者。图3-14表示了if语句运行时间的计算。

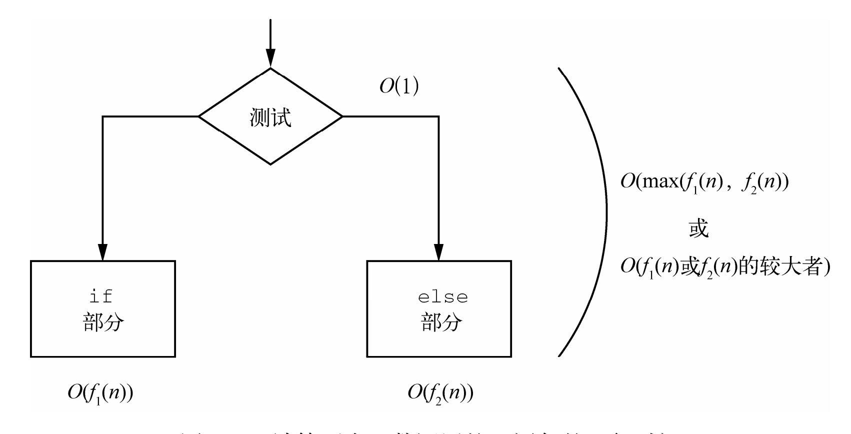


图3-14 计算不含函数调用的if语句的运行时间

(5) 程序块。如果程序块中各语句的运行时间上界分别是 $O\big(f_1(n)\big)$ 、$O\big(f_2(n)\big)$ 、…、$O\big(f_k(n)\big)$ ，那么整个程序块运行时间的上界就是 $O\big(f_1(n)+f_2(n)+\cdots+f_k(n)\big)$ 。如果可能的话，请使用求和规则简化这个表达式。程序块运行时间的计算规则如图3-15所示。

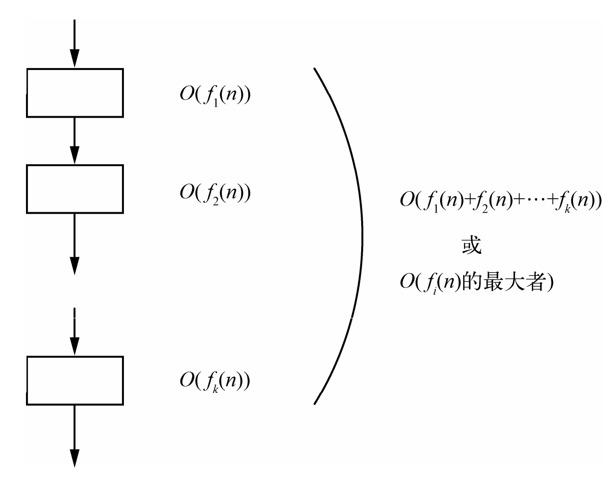


图3-15　计算不含函数调用的程序块的运行时间

可以应用这些规则，从较小的语句开始向上遍历表示复合语句构造的结构树。或者，可以将这些规则的应用视为从递归依据所涵盖的简单语句开始，逐步变成更大的符合语句，在每一步应用5种归纳规则中任意一种合适的规则。不管我们怎样看待计算运行时间上界的过程，都需要在分析过组成复合语句的所有语句之后，再对复合语句加以分析。

✦　示例 3.21

我们来重新审视一下图3-11中的排序程序，它的结构树如图3-12所示。首先，已知图3-12中树叶位置的每条赋值语句所花的时间为 $O(1)$ 。继续向树的上方行进，就会遇到第(4)行和第(5)行的`if`语句。从示例3.15中可以回想起这一复合语句所花的时间为 $O(1)$ 。

接下来随着向上遍历该树（或者说从较小语句向它们所围绕的较大语句行进），就必须分析第(3)行至第(5)行的`for`循环。示例3.14就是完成这一工作的，从中可得出运行时间为 $O(n-i-1)$ 。在这里，我们选择将运行时间表示为具有n和i两个变量的函数。这一选择给我们带来了一些计算上的困难，而正如接下来将要看到的，其实可以选择 $O(n)$ 这个更松散的上界。要以 $O(n-i-1)$ 作为边界，就必须从图3-11的第(1)行看出`i`从不可能有 $n-1$ 这么大。因此 $n-i-1$ 是严格大于0的，并主导 $O(1)$ 。所以，我们不需要在 $O(n-i-1)$ 之外加上初始化`for`循环的指标`j`所花的时间 $O(1)$ 。

现在到了第(2)行至第(8)行的程序块。正如示例3.17中所描述的，该程序块的运行时间是对应4条赋值语句的4个 $O(1)$ 的和，加上第(3)至第(5)行复合语句的 $O(n-i-1)$ 。根据求和规则，以及我们看出的 $i<n$ 的结论，可以舍弃这些 $O(1)$ ，留下 $O(n-i-1)$ 作为这个程序块的运行时间。

最后，必须考虑从第(1)行到第(8)行的这个`for`循环。该循环在3.6节中没有得到分析，不过我们可以运用归纳规则(3)。该规则需要循环体（也就是第(2)行至第(8)行）的运行时间上界。我们刚确定了该程序块的边界为 $O(n-i-1)$ ，这展现了之前从未见过的情形。尽管`i`在该程序块内是个常量，然而`i`是外层`for`循环的循环指标，会随着循环变化。因此，我们不能

将边界 $O(n-i-1)$ 视作该循环全部迭代的运行时间。好在从第(1)行可以看出i不会小于0，所以 $O(n-1)$ 是 $O(n-i-1)$ 的上界。此外，根据低阶项不产生影响的规则，可以将 $O(n-1)$ 简化为 $O(n)$ 。

接下来需要确定循环进行的次数。因为`i`是从0到 $n-2$ ，所以显然要循环 $n-1$ 次。用 $n-1$ 乘上 $O(n)$ ，便得到 $O(n^2-n)$ 。再次舍去低阶项，就得到 $O(n^2)$ 是整个选择排序程序的运行时间上界。也就是说，选择排序的运行时间具有二次的上界。该二次上界是可能存在的最紧上界了，因为可以证明，如果这些元素一开始是倒序排列的，那么选择排序就要进行 $n(n-1)/2$ 次比较。

一如我们将要看到的，可以为归并排序得出 $n\log n$ 的运行时间边界。在实践中，除了对那些小的n值之外，归并排序要比选择排序更高效。归并排序有时比选择排序慢的原因就在于，$O(n\log n)$ 的上界与选择排序的边界 $O(n^2)$ 相比，隐藏了一个更大的常数。真实的情况是一对交叉的曲线，如3.3节中的图3-2所示。

3.7.3 循环运行时间更精确的上界

我们已经说过，要评估循环的运行时间，需要找出适用于循环每一次迭代的统一边界。不过，对循环更为细致的分析要分开处理每次迭代，并为每次迭代的上界求和。从技术上讲，必须将递增循环指标（如果循环是`for`循环）和测试循环条件的时间包括在内，以防出现操作的时间能引起决定性变化的罕见情况。一般来讲，更加细致的分析并不会改变答案，虽然在一些不寻常的循环中大多数迭代只花费很少时间，而一次或几次迭代却占据大量运行时间（这会使这种循环每次迭代时间之和，要明显小于迭代次数乘上每次迭代可能花的最大时间的积）。

✦ 示例 3.22

我们要对选择排序的外层循环进行这种更精确的分析。尽管付出了额外的努力，可还是会得到二次的上界。正如示例3.21所示，当指标变量`i`的值为i时，外层循环此次迭代的时间为 $O(n-i-1)$ 。`i`的范围是0到 $n-2$ ，因此所有迭代所花时间的上界就是 $O\left(\sum_{i=0}^{n-2}(n-i-1)\right)$ 。这个和式中所有项形成了一个算术级数，所以可以利用公式“第一项和最后一项的平均数乘以项数”。该公式告诉我们：

$$\sum_{i=0}^{n-2}(n-i-1)=n(n-1)/2=0.5n^2-0.5n$$

忽略低阶项和常数因子，可以看到 $O(0.5n^2-0.5n)$ 与 $O(n^2)$ 是相同的。这样就再次得出了结论：选择排序具有二次的运行时间上界。

示例3.21中的简单分析与示例3.22中更细致分析的区别如图3-16所示。在示例3.21中，将任一次迭代可能花费的最大时间当作每次迭代的时间，因此得到了长方形的区域作为图3-11中`for`循环运行时间的边界。在示例3.22中，通过图中的对角线为每次迭代确定了运行时间边界，因为每次迭代的时间是随着i线性递减的。因此，可以得出该三角形的面积以作为对运行时间的估计。不过，众所周知，图中三角形的面积是长方形面积的一半。因为常数因子2与其他被大O表示法隐藏的常数因子一样会消失，所以这两个运行时间上界其实是一样的。

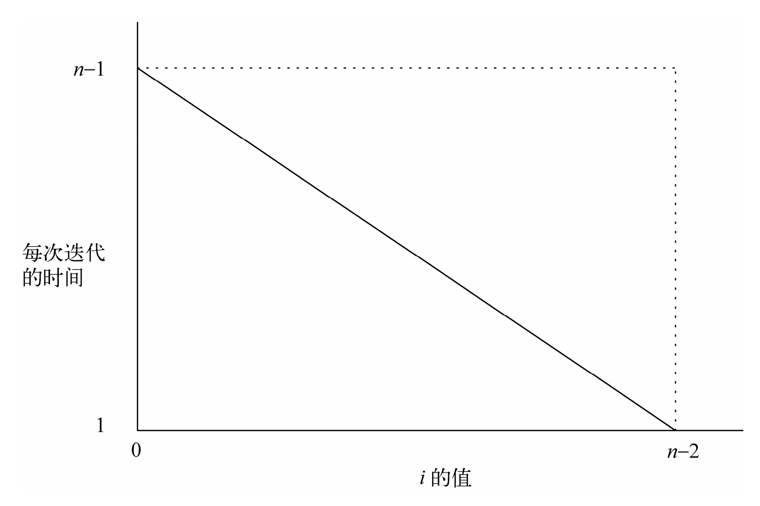


图3-16　对循环运行时间的简单估算和精确估算

3.7.4　习题

(1) 图3-17中的C语言程序会计算数组`A[0..n-1]`中各元素的平均值，并将最接近该平均值的元素的下标打印出来（若不止有一个这样的元素，则以先出现的为准）。假设 $n \geqslant 1$，而且不含对空数组的必要检测。画出结构树，展示这些语句是如何进一步组成更复杂的语句的，并给出该结构树中每一语句运行时间的简单大O上界和紧大O上界。整个程序的运行时间是多少？

```
        #include <stdio.h>
        #define MAX 100
        int A[MAX];

        main()
        {
            int closest, i, n;
            float avg, sum;

(1)         for (n = 0; n < MAX && scanf("%d", &A[n]) != EOF; n++)
(2)             ;
(3)         sum = 0;
(4)         for (i = 0; i < n; i++)
(5)             sum += A[i];
(6)         avg = sum/n;
(7)         closest = 0;
(8)         i = 1;
(9)         while (i < n) {
                /* 在下面的测试中为元素求平方，就不再
                   需要区分正数和负数的差异了。*/

(10)            if ((A[i]-avg)*(A[i]-avg) <
                  (A[closest]-avg)*(A[closest]-avg))
(11)                    closest = i;
(12)            i++;
            }
(13)        printf("%d\n",closest);
        }
```

图3-17　习题(1)的程序

(2) 图3-18所示的程序段会将 $n \times n$ 的矩阵A变形。画出该程序段的结构树，给出每一复合语句运行时间的大O上界。

(a) 用n和i的函数表示两个内层循环运行时间的边界。

(b) 用n的函数表示所有循环运行时间的边界。

对整个程序，你的答案和(a)、(b)部分之间是否存在大O差异？

```
(1)  for (i = 0; i < n-1; i++)
(2)      for (j = i+1; j < n; j++)
(3)          for (k = i; k < n; k++)
(4)              A[j][k] = A[j][k] - A[i][k]*A[j][i]/A[i][i];
```

图3-18　习题(2)的程序

(3) * 图3-19中的程序段对范围从1到n的整数i应用了示例3.8中讨论的"2的乘方"操作。画出该程序段的结构树，给出每一复合语句运行时间的大O上界。

(a) 用i的函数表示该while循环运行时间的边界。

(b) 用n的函数表示该while循环运行时间的边界。

对整个程序，你的答案和(a)、(b)部分之间是否存在大O差异？

```
(1)          for (i = 1; i <= n; i++) {
(2)              m = 0;
(3)              j = i;
(4)              while (j%2 == 0) {
(5)                  j = j/2;
(6)                  m++;
                 }
             }
```

图3-19　习题(3)的程序

(4) 图3-20中的函数会确定参数n是否为质数。请注意，如果n不是质数，它就可以被某个在2和 $\sqrt{n}$ 之间的整数i整除。画出该函数的结构树，用n的函数表示每一复合语句运行时间的大O上界。整个函数的运行时间又是多少？

```
        int prime(int n)
        {
            int i;

(1)         i = 2;
(2)         while (i*i <= n)
(3)             if (n%i == 0)
(4)                 return FALSE;
                else
(5)                 i++;
(6)         return TRUE;
        }
```

图3-20　习题(4)的程序

3.8 含函数调用的程序的分析

现在要展示的是如何分析包含函数调用的程序或程序段的运行时间。首先，如果所有的函数都是非递归的，可以从那些不调用其他函数的函数开始，每次确定一个组成该程序的函数的运行时间，然后为那些“只调用已确定运行时间的函数”的函数评估运行时间。我们以这种方式继续评估，直到评估完所有函数的运行时间。

不同函数可能有不同的输入大小的自然量度，这一事实带来了一些复杂性。在一般情况下，函数的输入就是该函数的参数列表。如果函数F调用了函数G，就必须将函数G中参数的大小量度与函数F所使用的大小量度联系起来。这里很难给出实用的通则，不过本节和下一节中的一些示例将有助于我们了解简单情况下为函数确定运行时间边界的过程是怎样的。

假设已经确定，函数F运行时间的良好上界是$O\big(h(n)\big)$，其中n是函数F参数大小的度量。那么在某条简单语句（比如一条赋值语句）中对F进行调用时，就要将$O\big(h(n)\big)$的开销加到那条语句的运行时间中。

当上界为$O\big(h(n)\big)$的函数出现在`while`语句、`do-while`语句或`if`语句的条件中，或出现在`for`语句的初始化、测试或重初始化中时，该函数调用的时间是按如下方法计算的。

(1) 如果函数调用是在`while`循环或`do-while`循环的条件中，或在`for`循环的条件或重初始化中，那么就要在每次迭代的时间边界上加上$h(n)$，然后按照3.7节中获取循环运行时间的方式继续下去。

(2) 如果函数调用是在`for`循环的初始化中，就在循环的时间开销上加上$O\big(h(n)\big)$。

(3) 如果函数调用是在`if`语句的条件中，就在该语句的时间开销上加上$h(n)$。

简述程序分析

大家应该从3.7节和3.8节中了解到的主要观点如下。

- 一系列语句的运行时间就是每一条语句运行时间的和。通常，如果某一语句的运行时间至少与其他语句一样大，那么它就可以主导其他语句。根据求和规则，主导语句的运行时间就是这一系列语句的大O运行时间。
- 要计算循环的运行时间，先要将循环体的时间与各控制步骤（比如重初始化`for`循环的循环指标并将其与上限相比较）的运行时间相加。用这个时间去乘以循环迭代次数的上界。接着，将那些一次性完成的步骤（比如初始化或第一次终止测试）的时间加上，以防循环迭代0次的情况出现。
- 选择语句（例如`if-else`语句）的运行时间是决定执行哪个分支所花的时间与各分支运行时间中较大的那个相加而得到的之和。

✦ 示例 3.23

让我们分析一下图3-21中的（无意义的）程序。首先，你会注意到这不是一个递归程序。`main`函数会调用`foo`函数和`bar`函数，而且`foo`函数会调用`bar`函数，不过这就是全部的调用关系了。图3-22所示的图称为调用图，表示函数调用其他函数的方式。因为图中不含循环，所以

程序中没有递归调用，而且可以首先从“第0组”（就是不调用其他函数的函数，在本例中就是bar函数）开始分析这些函数，接着处理“第1组”（就是只调用第0组中函数的函数，在本例中就是foo函数），再处理“第2组”（就是只调用第0组和第1组中函数的函数，在本例中就是main函数）。至此，工作就完成了，因为所有的函数都已经被分组了。在一般情况下，可能要考虑分更多的组，不过只要其中不含循环，最终就能将每个函数都放在一个组别中。

```
        #include <stdio.h>
        int bar(int x, int n);
        int foo(int x, int n);

        main()
        {
            int a, n;

(1)         scanf("%d", &n);
(2)         a = foo(0,n);
(3)         printf("%d\n", bar(a,n));
        }

        int bar(int x, int n)
        {
            int i;

(4)         for (i = 1; i <= n; i++)
(5)             x += i;
(6)         return x;
        }

        int foo(int x, int n)
        {
            int i;

(7)         for (i = 1; i <= n; i++)
(8)             x += bar(i,n);
(9)         return x;
        }
```

图3-21　展示非递归函数调用的程序

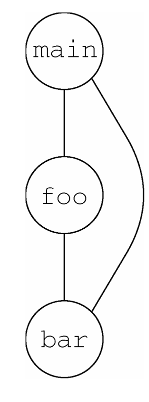


图3-22　图3-21所示程序的调用图

我们分析函数运行时间的顺序，也就是为了理解该程序的行为而对其进行研究的顺序。因此，首先考虑一下bar函数是做什么的。第(4)行和第(5)行的for循环会将从1到n的这n个整数都加到x上，结果就是$bar(x,n)$等于$x+\sum_{i=1}^{n} i$。这里的和式$\sum_{i=1}^{n} i$又是个为算术级数求和的例子，只要将第一项与最后一项相加，乘以项数，然后再除以2即可。也就是$\sum_{i=1}^{n} i=(1+n)n/2$。因此，$bar(x,n)=x+(1+n)n/2$。

现在，考虑一下foo函数，它会给它的参数x加上和式

$$\sum_{i=1}^{n} bar(i,n)$$

根据我们对bar函数的了解可知，$bar(i,n)=i+n(n+1)/2$。因此，foo函数就是给x加上了$\sum_{i=1}^{n}\big(i+n(n+1)/2\big)$这个量。这样就要为另一个算术级数求和了，而这个算术级数需要更多的代数变换。不过，读者可以验证一下，foo函数加到x上的这个量就是$(n^3+2n^2+n)/2$。

最后看看main函数。我们在第(1)行读入n，在第(2)行将foo应用到0和n上。根据我们对foo函数的理解，第(2)行foo(0,n)的值就是0加上$(n^3+2n^2+n)/2$。在第(3)行，要将bar(foo(0,n),n)的值打印出来，根据我们对bar函数的理解，这就是$n(n+1)/2$与foo(a,n)当前值的和。因此，要打印的值就是$(n^3+2n^2+n)/2$。

现在来分析图3-21所示程序的运行时间，从bar函数开始，到foo函数，再到main函数，一如我们在示例3.23中所做的那样。在这种情况下，我们要确定值n是所有三个函数的输入的大小。也就是说，即便我们通常想考虑函数所有参数的“大小”，但在本例中函数的运行时间只取决于n。

要分析bar函数，先要注意到第(5)行所花的时间为$O(1)$。第(4)行和第(5)行的for循环要迭代n次，所以第(4)行和第(5)行的运行时间是$O(n)$。第(6)行花的时间也是$O(1)$，所以第(4)行至第(6)行的程序块的运行时间是$O(n)$。

接着分析foo函数。第(8)行的赋值语句花的时间是$O(1)$加上调用bar(i,n)所用的时间。而我们已经知道，该调用花的时间为$O(n)$，所以第(8)行的运行时间就是$O(n)$。第(7)行和第(8)行的for循环要迭代n次，所以可以用循环体的运行时间$O(n)$乘上循环迭代的次数n，得到调用foo函数的运行时间是$O(n^2)$。

最后来分析main函数。第(1)行所花的时间为$O(1)$，第(2)行对foo函数的调用所花的时间为$O(n^2)$，第(3)行的打印语句所花的时间为$O(1)$加上调用bar函数所花的时间。而后者所花时间为$O(n)$，所以整个第(3)行所花的时间为$O(1)+O(n)$。因此从第(1)行到第(3)行的整个程序块的运行时间为$O(1)+O(n^2)+O(1)+O(n)$。根据求和规则，可以消除第二项之外的所有项，得出该函数的运行时间为$O(n^2)$。也就是说，第(2)行对foo函数的调用决定了整个时间开销。

证明和对程序的理解

读者可能注意到，在对图3-21所示程序的研究中，我们能理解程序在做什么，却不能像在第2章中那样正式地证明点什么。不过，在这表面之下却潜藏着诸多简单的归纳证明。例如，需要对第(4)行和第(5)行循环迭代的次数进行归纳，证明在我们用值为i的i开始迭代之前，x的值是x的初始值加上$\sum_{j=1}^{i-1} j$。请注意，如果$i=1$，这个和式不含任何项，则其值会为0。

习题

(1) 证明示例3.23中的结论：$\sum_{i=1}^{n}(i+n(n+1)/2)=(n^3+2n^2+n)/2$。

(2) 假设prime(n)是运行时间为$O(\sqrt{n})$的函数调用。考虑一下函数体如下的函数：

```
if ( prime(n) )
    A;
else
    B;
```

分别假设：

(a) A所花的时间为$O(n)$，B所花的时间为$O(1)$；

(b) A和B所花的时间都为$O(1)$。

用n的函数表示出这两种情况下该函数运行时间的简单大O上界和紧大O上界。

(3) 考虑函数体如下的函数：

```
sum = 0;
for (i = 1; i <= f(n); i++)
    sum += i;
```

其中$f(n)$是函数调用。分别假设：

(a) $f(n)$的运行时间是$O(n)$，而$f(n)$的值是$n!$；

(b) $f(n)$的运行时间是$O(n)$，而$f(n)$的值是n；

(c) $f(n)$的运行时间是$O(n^2)$，而$f(n)$的值是n；

(d) $f(n)$的运行时间是$O(1)$，而$f(n)$的值是0。

用n的函数表示出这4种情况下该函数运行时间的简单大O上界和紧大O上界。

(4) 绘出2.8节归并排序程序中函数的调用图。那个程序是否为递归程序？

(5) * 假设图3-21中foo函数的第(7)行被替换为

```
for (i = 1; i <= bar(n,n); i++)
```

那么main函数的运行时间会是多少？

3.9 递归函数的分析

确定递归调用自身的函数的运行时间，需要比分析那些非递归函数耗费更多的精力。递归函数的分析需要我们将程序中的每个函数F与某个未知的运行时间$T_F(n)$关联起来。这一未知的函数将F的运行时间表示为F函数参数的大小n的函数。然后构建一套归纳定义，称为$T_F(n)$的递推关系，将$T_F(n)$与同一程序中其他函数G及其相应的参数大小k表示的$T_G(k)$形式关联起来。如果F是直接递归的，那么G中至少有一个将与F是相同的。

$T_F(n)$的值通常是通过对参数大小n的归纳取得的。因此，需要选择合适的参数大小，保证随着递归的进行，函数在被调用时所使用的参数在逐渐减小。这一要求与我们在2.9节中试图证明有关递归程序的命题时遇到的要求别无二致。这应该没什么可奇怪的，因为有关程序运行时间的命题正是我们可能试着证明的与程序相关的某种内容。

一旦找到了合适的参数大小，就可以考虑以下两种情况了。

(1) 参数大小足够小，使F不进行递归调用。这种情况对应$T_F(n)$归纳定义中的依据。

(2) 对于较大的参数大小，将至少会发生一次递归调用。请注意，无论F进行怎样的递归调用，不管是对其自身还是对某个其他函数G进行递归调用，都只可能使用更小的参数。

这种情况对应$T_F(n)$归纳定义中的归纳步骤。

通过对函数F的代码的研究，并完成如下操作，可以得出$T_F(n)$递推关系的定义。

(a) 对函数G的每次调用或表达式中函数G的每次使用（请注意，G可能就是F），用$T_G(k)$表示该次调用的运行时间，其中k是对该次调用中参数大小的合理度量。

(b) 运用前面几节中介绍的技巧评估函数F的函数体的运行时间，不过要将$T_G(k)$这样的项留作未知函数，而不是诸如n^2这样的具体函数。一般不能用求和规则这样的简化技巧把这些项与具体函数结合起来。我们必须对F进行两次分析，一次假设F的参数大小n足够小，使得函数未进行递归调用，而另一次假设n不是那么小。因此，我们得到了两个表示F函数运行时间的表达式。其一（依据表达式）是$T_F(n)$递推关系的依据，另一个（归纳表达式）则是$T_F(n)$递推关系的归纳部分。

(c) 在得出的有关函数F运行时间的依据表达式和归纳表达式中，用特定常数乘上有关函数（例如$cf(n)$）的形式来代替像$O\big(f(n)\big)$这样的大O项。

(d) 如果输入大小的依据值为a，令$T_F(a)$是在假设不存在递归调用的情况下，由步骤(c)得出的依据表达式。还有，令$T_F(n)$是从步骤(c)得到的n值不为依据值a的情况下的归纳表达式。

```
            int fact(int n)
            {
(1)             if (n <= 1)
(2)                 return 1; /* 依据 */
                else
(3)                 return n*fact(n-1); /* 归纳 */
            }
```

图3-23　计算$n!$的程序

通过求解这个递推关系，就可以确定整个函数的运行时间。在3.11节中，我们将介绍一些一般性的技巧，用来在对普通递归函数的分析中求解这种递推关系。而现在，我们要通过特别手段来求解这些递推关系。

✦ 示例 3.24

我们来重新考虑一下2.7节中计算阶乘函数的递归程序。因为只涉及`fact`这一个函数，所以使用$T(n)$表示该函数未知的运行时间。我们将使用参数的值n作为参数的大小。显然，当参数为n时进行的`fact`函数的递归调用，要使用更小的参数，准确地说是$n-1$。

我们选择$n=1$作为$T(n)$归纳定义的依据，因为当`fact`函数的参数为1时，它不执行任何递归调用。当$n=1$时，第(1)行的条件为真，因此对`fact`的调用会执行第(1)行和第(2)行。每一行花的时间都是$O(1)$，所以依据情况中`fact`的运行时间为$O(1)$。也就是说$T(1)$是$O(1)$。

现在考虑当$n>1$时会发生什么。第(1)行的条件为假，因此只执行第(1)行和第(3)行。第(1)行花的时间是$O(1)$，而第(3)行会在乘法和赋值上用掉$O(1)$，并在对`fact`的递归调用上花费$T(n-1)$。也就是说，当$n>1$时，`fact`的运行时间是$O(1)+T(n-1)$。因此可以用以下递推关系定义$T(n)$。

依据。$T(1)=O(1)$。

归纳。对$n>1$，$T(n)=O(1)+T(n-1)$。

现在要引入一些常数符号来表示隐藏在各大O表达式中的常数，就像之前在规则(c)中表述

的那样。在这种情况下，可以用某个常数a代替依据中的$O(1)$，并用某个常数b替代归纳中的$O(1)$。这些变化给了我们如下的递推关系。

依据。$T(1)=a$。

归纳。对$n>1$，$T(n)=b+T(n-1)$。

现在必须求解$T(n)$的这一递推关系。我们很容易计算出靠前的一些值，由递推依据$T(1)=a$，以及归纳规则，我们得到

$$T(2)=b+T(1)=a+b$$

继续使用归纳规则，就得到

$$T(3)=b+T(2)=b+(a+b)=a+2b$$

然后是

$$T(4)=b+T(3)=b+(a+2b)=a+3b$$

至此，不难猜测，对所有的$n\geqslant 1$，有$T(n)=a+(n-1)b$。其实，计算一些样本值，接着猜测解决方案，并最终通过归纳法证明猜测正确，这就是我们常用来处理递推关系的方法。

不过，在这个例子中，我们可以使用反复代换（repeated substitution）的方法直接得出解决方案。首先，在递归等式中进行如下变量代换，用m替换n，就得到

对$m>1$，$T(m)=b+T(m-1)$ （3.3）

现在，可以用n、$n-1$、$n-2$、…、2替换等式(3.3)中的m，得到一系列的等式

$$\begin{aligned}
(1)\ & T(n) &&= b+T(n-1)\\
(2)\ & T(n-1) &&= b+T(n-2)\\
(3)\ & T(n-2) &&= b+T(n-3)\\
& \cdots\\
n-1)\ & T(2) &&= b+T(1)
\end{aligned}$$

接下来，可以利用上述系列等式中的第(2)行，来替换第(1)行中的$T(n-1)$，从而得到等式

$$T(n)=b+\big(b+T(n-2)\big)=2b+T(n-2)$$

现在用第(3)行替换上式中的$T(n-2)$，就得到

$$T(n)=2b+\big(b+T(n-3)\big)=3b+T(n-3)$$

按这种方式继续下去，每一次都将$T(n)-i$替换为$b+T(n-i-1)$，直到向下达到$T(1)$。至此，就得到了等式

$$T(n)=(n-1)\,b+T(1)$$

接着可以利用依据，用a替换$T(1)$，就可以得到$T(n)=a+(n-1)b$。

如果想让该分析过程更正式，就需要通过归纳法，对我们在反复对$T(n-i)$进行替换时的直观观察结果加以证明。因此，我们要通过对i的归纳证明如下命题。

命题$S(i)$。如果$1\leqslant i\leqslant n$，那么$T(n)=ib+T(n-i)$。

依据。依据为$i=1$，$S(1)$是说$T(n)=b+T(n-1)$。这是对$T(n)$的定义中的归纳部分，因此已知为真。

归纳。如果$i\geqslant n-1$，就没什么要证明的，因为命题$S(i+1)$的开头是“如果$1\leqslant i+1\leqslant n$”，而当`if`语句的条件为假时，不管“那么”后面是如何表述的，该命题都为真。在这种情况下，若$i\geqslant n-1$，则条件$i+1<n$一定为假。所以$S(i+1)$一定为真。

难点就在于，当$i\leqslant n-2$的时候。在这种情况下，$S(i)$就是$T(n)=ib+T(n-i)$。因为

$i \leqslant n-2$，所以$T(n-i)$的参数至少为2。因此可以将该归纳规则应用到T上，也就是用$n-i$替换等式(3.3)中的m，从而得出等式$T(n-i)=b+T(n-i-1)$。当我们用$b+T(n-i-1)$替换掉等式$T(n)=ib+T(n-i)$中的$T(n-i)$时，就得到$T(n)=ib+\big(b+T(n-i-1)\big)$，重组这些项就得到

$$T(n)=(i+1)b+T\big(n-(i+1)\big)$$

这个等式就是命题$S(i+1)$，而且我们现在已经证明了归纳步骤。

现在已经证明了$T(n)=a+(n-1)b$。不过，a和b都是未知的常数。因此，这样表示解决方案是不行的。不过，可以将$T(n)$表示为n的多项式，即$bn+(a-b)$，接着再用大O表达式来替代这些项，就得到了$O(n)+O(1)$。利用求和规则，还可以消掉$O(1)$，从而得出$T(n)$是$O(n)$。这就有意义了，它表示：要想计算$n!$，就要利用对`fact`的n次（实际调用次数刚好为n）调用的顺序，其中每次调用所需时间为$O(1)$，不计入花在执行对`fact`的递归调用上的时间。

习题

(1) 为2.9节习题(2)中提到的`sum`函数（它是作为程序输入的表的长度的函数）的运行时间建立递推关系。请用（未知的）常数替换大O项，并试着求解这种递推关系。`sum`的运行时间是多少？

(2) 对2.9节习题(3)中提到的`find0`函数重复习题(1)中的练习。合适的大小量度是什么？

(3) * 对2.7节中图2-22所示的选择排序程序重复习题(1)中的练习。合适的大小量度是什么？

(4) ** 对图3-24中的函数重复习题(1)中的练习，该函数是计算斐波那契数的（最开始的两个数是1，之后的每个数都是其前两个相邻数字之和。前7个斐波那契数分别是1、1、2、3、5、8、13）。请注意，n的值是合适的参数大小，而且大家需要使用1和2作为依据情况。

```
int fibonacci(int n)
{
    if (n <= 2)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

图3-24　计算斐波那契数的C语言函数

(5) * 编写递归程序计算`gcd(i,j)`，就是两个整数i和j的最大公约数，如2.7节习题(8)中概述的那样。证明该程序的运行时间是$O(\log i)$。提示：在我们调用`gcd(m,n)`两次后证明这一点（其中$m \leqslant i/2$）。

3.10　归并排序的分析

我们现在要分析2.8节中介绍过的归并排序算法。首先要证明，`merge`函数和`split`函数在处理长度为n的表时，所花的时间都是$O(n)$；接着使用这些边界来证明`MergeSort`函数在处理长度为n的表时所花的时间为$O(n\log n)$。

3.10.1　merge函数的分析

首先分析递归函数`merge`，我们在图3-25中再次展示了它的代码。`merge`函数参数大小n的合适概念是表`list1`和`list2`的长度之和。因此，设$T(n)$是当参数表的长度之和为n时`merge`

函数所花的时间。我们可以拿 $n=1$ 的情况作为依据情况，因此必须在list1和list2二者中有一个为空而另一个仅含一个元素的假设下对图3-25进行分析。有以下两种情况。

(1) 如果第(1)行的测试（也就是list1等于NULL的测试）成功，我们就返回list2，这所花的时间为 $O(1)$ 。第(2)行至第(7)行就不会执行。因此，整个函数调用所花的时间为测试第(1)行选择的 $O(1)$ 和执行第(1)行赋值的 $O(1)$ ，总共是 $O(1)$ 。

(2) 如果第(1)行的测试失败，就说明list1不为空。因为我们假设两个表的长度之和只是1，所以list2一定为空。因此，第(2)行的测试（即list2等于NULL的测试）一定会成功。那么我们就要花 $O(1)$ 来执行第(1)行的测试，花 $O(1)$ 执行第(2)行的测试，再花 $O(1)$ 在第(2)行返回list1。第(3)行至第(7)行不会执行。所花的时间还是 $O(1)$ 。

这样可以得出在依据情况中merge的运行时间为 $O(1)$ 。

```
    LIST merge(LIST list1, LIST list2)
    {
(1)     if (list1 == NULL) return list2;
(2)     else if (list2 == NULL) return list1;
(3)     else if (list1->element <= list2->element) {
(4)         list1->next = merge(list1->next, list2);
(5)         return list1;
        }
        else { /* list2 的第一个元素更小 */
(6)         list2->next = merge(list1, list2->next);
(7)         return list2;
        }
    }
```

图3-25 merge函数

现在来考虑归纳情况，也就是表长度之和大于1的情况。当然，即便长度之和为2或者更大，仍然可能有一个表为空。因此，嵌套的选择语句所表示的4种情况都可能发生。图3-25中程序的结构树如图3-26所示。我们可以从结构树的底部开始，向上分析该程序。

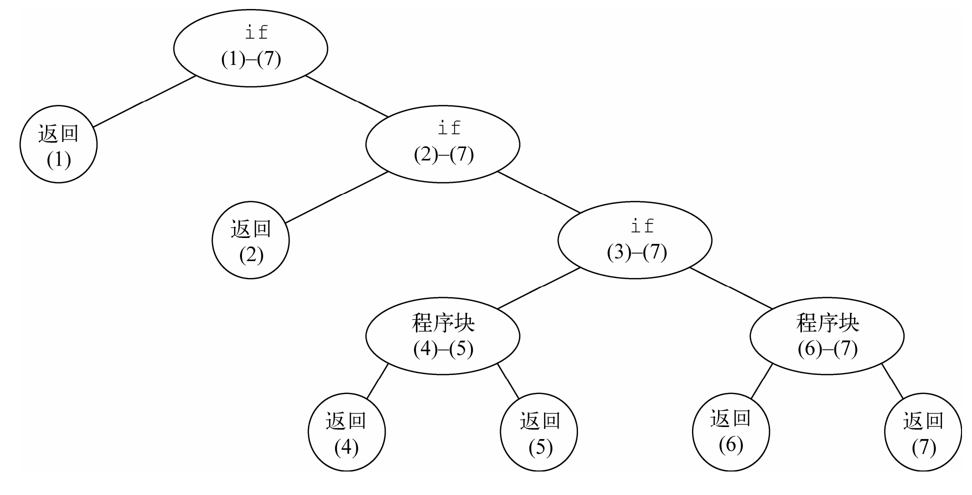


图3-26 merge的结构

最内层的选择是从第(3)行的“if”开始的，我们在那里会测试哪个表的第一个元素更小，

并根据测试的结果选择执行第(4)行和第(5)行，或执行第(6)行和第(7)行。第(3)行的条件需要花 $O(1)$ 的时间来评估，第(5)行要花上 $O(1)$ 来评估，而第(4)行所花的时间是 $O(1)$ 加上递归调用merge所花的时间 $T(n-1)$。请注意，$n-1$ 是递归调用的参数大小，因为我们已经从一个表中剔除了一个元素，并保持另一个表不变。因此第(4)行和第(5)行的程序块所花的时间为 $O(1)+T(n-1)$。

对第(6)和第(7)行中else部分的分析是完全一样的：第(7)行所花时间为 $O(1)$，而第(6)行所花时间为 $O(1)+T(n-1)$。因此，在选取if部分和else部分运行时间的最大值时，会发现这两者其实是相同的。测试条件所花的时间 $O(1)$ 可以忽略，因此可以得出结论：最内层选择的运行时间是 $O(1)+T(n-1)$。

现在继续分析从第(2)行开始的选择。我们要在这一行测试list2是否等于NULL。测试条件的时间为 $O(1)$，而if 部分的时间（就是第(2)行的返回）也是 $O(1)$。不过，else部分是第(3)行至第(7)行的选择语句，这部分语句的运行时间我们刚才确定过了，是$O(1)+T(n-1)$。因此，第(2)行至第(7)行的选择所花的时间为

$$O(1)+\max\big(O(1),O(1)+T(n-1)\big)$$

最大值中的第二项主导了第一项，也主导了测试条件所花的时间 $O(1)$。因此，从第(2)行开始的if语句的运行时间也是 $O(1)+T(n-1)$。

最后，要对最外层的if语句进行同样的分析。从根本上讲，对时间起主导作用的还是由第(2)行至第(7)行组成的else部分的时间。

递归的共通形式

很多极简单的递归函数（比如fact和merge）都会执行一些所需时间为 $O(1)$ 的操作，然后对它们自己执行参数大小减小1的递归调用。假设依据情况花的时间为 $O(1)$，可以看到这样的函数总能形成 $T(n)=O(1)+T(n-1)$ 这样的递推关系。$T(n)$ 的解是 $O(n)$，或者是参数大小的线性关系。在3.11节中我们还将看到对这一原则的一些概括。

也就是说，这些含递归调用的情况（比如第(4)行和第(5)行，或第(6)行和第(7)行）下的时间，主导了不含递归调用情况下的时间，还主导了第(1)、(2)和(3)行中所有3次测试的时间。因此，当$n>1$时，merge函数的运行时间上界就是 $O(1)+T(n-1)$。因此可以得到用于定义 $T(n)$ 的如下递推关系。

依据。$T(1)=O(1)$。

归纳。对 $n>1$，$T(n)=O(1)+T(n-1)$。

这些等式与示例3.24中为fact函数得出的那些等式如出一辙。因此，求解过程是相同的，可以得出 $T(n)$ 是 $O(n)$ 的结论。该结果从直观上讲是成立的，因为merge函数的工作原理就是花 $O(1)$ 的时间从其中一个表里删除一个元素，然后对剩余的表递归地调用自身。这种递归调用遵循着递归调用的次数不大于表长度之和的原则。如果不考虑其递归调用所花时间，那么每次调用均耗时 $O(1)$，如此就可以得出merge的运行时间将会是 $O(n)$。

3.10.2　split函数的分析

现在来考虑一下split函数，我们在图3-27中再次展示了该函数。对split函数的分析和对merge函数的分析非常相似。我们令表的长度为参数的大小n，而且这里使用 $T(n)$ 表示split函数处理长度为n的表所花的时间。

```
     LIST split(LIST list)
     {
         LIST pSecondCell;

(1)      if (list == NULL) return NULL;
(2)      else if (list->next == NULL) return NULL;
         else { /* 至少有两个单元 */
(3)          pSecondCell = list->next;
(4)          list->next = pSecondCell->next;
(5)          pSecondCell->next = split(pSecondCell->next);
(6)          return pSecondCell;
         }
     }
```

图3-27 split函数

我们选取 $n=0$ 和 $n=1$ 作为依据。如果 $n=0$，也就是说表为空，那么第(1)行的测试就会成功，而我们将在第(1)行返回NULL。第(2)行至第(6)行就不会执行。因此所花的时间为 $O(1)$。如果 $n=1$，也就是表只含一个元素，那么第(1)行的测试失败，不过第(2)行的测试成功。因此我们会在第(2)行返回NULL，而且不执行第(3)行至第(6)行。同样，这两条测试语句和一条返回语句只需要 $O(1)$ 的时间。

接着考虑 $n>1$ 时的归纳部分，这里存在3条选择分支，类似我们在分析merge函数时遇到的4条分支。简单来说，可以看出，第(1)行和第(2)行的测试不论是执行一个还是两者都执行，所花时间都是 $O(1)$，正如我们最终为merge函数得出的结论那样。而且，如果这两个测试中有一个为真，就会致使我们在第(1)行或第(2)行返回的情况中，多花的时间也是 $O(1)$。占主导的时间是两个测试均失败的情况，也就是表长度至少为2的情况。在这种情况下，第(3)行至第(6)行的语句都要执行。除了第(5)行的递归调用外，其他的内容所花的时间为 $O(1)$。而递归调用的时间是 $T(n-2)$，因为该参数表是list原来的值减去它的前两个元素（想知道原因，可以参考2.8节中的内容，特别是图2-28）。因此，归纳情况下的 $T(n)$ 是 $O(1)+T(n-2)$。

可以建立如下递推关系。

依据。$T(0)=O(1)$，且 $T(1)=O(1)$。

归纳。对 $n>1$，$T(n)=O(1)+T(n-2)$。

如示例3.24所述，接下来必须引入某些常数来表示隐藏在 $O(1)$ 背后的比例常数。可以分别用常数a和b表示依据中 $T(0)$ 和 $T(1)$ 的 $O(1)$，并用常数c表示归纳步骤中的 $O(1)$。因此，可以将上述递归定义重写为

依据。$T(0)=a$，且 $T(1)=b$。

归纳。对 $n\geqslant 2$，$T(n)=c+T(n-2)$。

我们先来求一下 $T(n)$ 的前几个值。由依据，显然有 $T(0)=a$和 $T(1)=b$。可以使用归纳步骤得出

$$
\begin{aligned}
T(2) &= c+T(0)=a+c \\
T(3) &= c+T(1)=b+c \\
T(4) &= c+T(2)=c+(a+c)=a+2c \\
T(5) &= c+T(3)=c+(b+c)=b+2c \\
T(6) &= c+T(4)=c+(a+2c)=a+3c
\end{aligned}
$$

对 $T(n)$ 的计算其实是两部分单独的计算，一部分是n为奇数的情况，一部分是n为偶数的情况。对偶数n，我们有 $T(n)=a+cn/2$ 。这是可行的，因为对长度为偶数的表，剔除两个元素所花的时间为c，而且在经过 $n/2$ 次递归调用后，就会得到不再对其进行递归调用而且所花时间为a的空表。

对奇数长度的表来说，还是要花时间c来剔除两个元素的。在经过 $(n-1)/2$ 次调用后，我们得到了一个长度为1的表，而且需要的时间为b。因此，奇数长度的表所需的时间将是 $b+c(n-1)/2$ 。

对这些观察结果的归纳证明与示例3.24中的证明过程非常近似，就是要证明如下命题。

命题 $S(i)$。如果 $1 \leqslant i \leqslant n/2$ ，那么 $T(n)=ic+T(n-2i)$ 。

在该命题的证明过程中，我们使用$T(n)$定义中的归纳规则，用参数m将其重写为

对
$$m \geqslant 2\text{，}\quad T(m)=c+T(m-2) \tag{3.4}$$
接着就可以按照如下方式用归纳法证明 $S(i)$ 了。

依据。依据是 $i=1$ ，也就是用n替代m后的等式(3.4)。

归纳。因为 $S(i)$ 是“如果……那么……”的形式，所以若 $i \geqslant n/2$ ，则 $S(i+1)$ 恒为真。因此，若 $i \geqslant n/2$ ，我们就不需要对归纳步骤（即由 $S(i)$ 可得到 $S(i+1)$ ）加以证明。

难点是当 $1 \leqslant i \leqslant n/2$ 时。在这种情况下，假定归纳假设$S(i)$为真，即 $T(n)=ic+T(n-2i)$ 。用 $n-2i$ 替换(3.4)中的m，就得到
$$T(n-2i)=c+T(n-2i-2)$$
如果替换 $S(i)$ 中的 $T(n-2i)$ ，就得到
$$T(n)=ic+\big(c+T(n-2i-2)\big)$$
如果对等式右边的项加以组合，就有
$$T(n)=(i+1)c+T\big(n-2(i+1)\big)$$

这就是命题 $S(i+1)$。因此我们证明了归纳步骤，而且得出 $T(n)=ic+T(n-2i)$ 。

现在，若n为偶数，则令 $i=n/2$ 。则 $S(n/2)$ 就表示 $T(n)=cn/2+T(0)$ ，也就是 $a+cn/2$ 。如果n为奇数，就令 $i=(n-1)/2$ 。 $S\big((n-1)/2\big)$ 就表示 $T(n)=c(n-1)/2+T(1)$ ，也就等于 $b+c(n-1)/2$ ，因为有 $T(1)=b$ 。

最后，必须将特定于编译器和机器的常数a、b和c改写为大O表示。多项式 $a+cn/2$ 和 $b+c(n-1)/2$ 都具有与n成比例的高阶项。因此，该问题中不论n是奇数还是偶数其实是没关系的，两种情况下`split`的运行时间都是 $O(n)$ 。这又是个很直观的正确解答，因为对长度为n的表来说，`split`会进行约 $n/2$ 次递归调用，每次调用的时间都是 $O(1)$ 。

3.10.3　`MergeSort`函数

最后要介绍一下`MergeSort`函数，我们在图3-28中再次展示了该函数。对参数大小的合适量度n还是待排序表的长度。在这里，我们要使用 $T(n)$ 表示`MergeSort`处理长度为n的表的运行时间。

我们选取 $n=1$ 的情况作为依据情况，而 $n>1$（发生递归调用）的情况则作为归纳情况。如果对`MergeSort`加以研究，就会发现，除非从另一个函数中调用参数为空表的`MergeSort`，不然是没办法在参数为空表的情况下进行调用的。原因在于，只有当表中至少具有两个元素时也就是分拆后得到的两个表中都至少有一个元素时，才会执行第(4)行。因此可以忽略 $n=0$ 的情况，并直接从 $n=1$ 开始进行归纳证明。

```
     LIST MergeSort(LIST list)
     {
         LIST SecondList;

(1)      if (list == NULL) return NULL;
(2)      else if (list->next == NULL) return list;
         else {
             /* 表中至少有两个元素 */
(3)          SecondList = split(list);
(4)          return merge(MergeSort(list), MergeSort(SecondList));
         }
     }
```

图3-28 归并排序算法

依据。如果list由一个元素构成，就会执行第(1)行和第(2)行，而不执行其他代码。因此，在依据情况中，$T(1)$ 是 $O(1)$ 。

归纳。在归纳情况中，第(1)行和第(2)行的测试都是失败的，因此可以执行第(3)行和第(4)行的程序块。为了简化问题，可以假设n是2的乘方。作出这种假设的好处在于，当n为偶数时，刚好会将表分割成长度为 $n/2$ 的两等分。此外，如果n是2的乘方，那么 $n/2$ 也是2的乘方，每次递归结束二分出来的都是等分的表，直到每个表中只含一个元素为止。当 n>1 时，MergeSort所花的时间为下列各项之和。

(1) 两次测试所花的 $O(1)$ 。

(2) 第(3)行的赋值和对split的调用所花的 $O(1)+O(n)$ 。

(3) 第(4)行对MergeSort第1次递归调用所花的 $T(n/2)$。

(4) 第(4)行对MergeSort第2次递归调用所花的 $T(n/2)$。

(5) 第(4)行调用merge所花的 $O(n)$ 。

(6) 第(4)行的返回语句所花的 $O(1)$ 。

跳过某些值的归纳法

读者不应该为MergeSort函数的分析中涉及的新型归纳法感到担心，尽管在证明过程中我们跳过了除2的乘方之外的所有数值。一般情况下，如果i_1、i_2、…是一列与我们想证明的命题S有关的整数，就可以证明 $S(i_1)$ 作为依据，并对所有的j证明，$S(i_j)$ 可推出 $S(i_{j+1})$。这就是一般情况下我们所认为的对j进行归纳的归纳证明。更精确地说，由 $S'(j)=S(i_j)$ 定义命题S'。然后通过对j的归纳证明 $S'(j)$ 。这样的话，就可以是 $i_1=1$ 、$i_2=2$ 、$i_3=4$ ，而一般形式就是 $i_j=2^{j-1}$ 。

顺便提一句，请注意，MergeSort的运行时间 $T(n)$ 不会随着n的增加而减少。因此，证明了对等于2的乘方的n有 $T(n)$ 是 $O(n\log n)$ ，也就证明了对所有的n都有 $T(n)$ 是 $O(n\log n)$ 。

如果将这些项加起来，然后依据调用split和merge的 $O(n)$ 更大而舍弃 $O(1)$ ，就可以得出在归纳情况中，MergeSort的运行时间边界是 $2T(n/2)+O(n)$ 。因此得到以下递推关系。

依据。$T(1)=O(1)$ 。

归纳。$T(n)=2T(n/2)+O(n)$ ，其中 n是2的乘方而且大于1。

下一步是要用含具体常数的函数代替大O表达式。我们在依据中用常数a代替 $O(1)$ ，并在归

纳步骤中用bn代替$O(n)$，因此递推关系就变形为

依据。$T(1)=a$。

归纳。$T(n)=2T(n/2)+bn$，其中n是2的乘方而且大于1。

这一递推关系要比我们之前了解的更难，不过我们还是可以利用相同的技巧。首先，可以为一些较小的n值直接写出$O(n)$的值。依据说明了$T(1)=a$，而归纳步骤则告诉我们

$$
\begin{aligned}
T(2) &= 2T(1)+2b & & = 2a+2b \\
T(4) &= 2T(2)+4b &= 2(2a+2b)+4b & = 4a+8b \\
T(8) &= 2T(4)+8b &= 2(4a+8b)+8b & = 8a+24b \\
T(16) &= 2T(8)+16b &= 2(8a+24b)+16b & = 16a+64b
\end{aligned}
$$

想直接看出接下来的情况可不容易。显然，a的系数与n的值是同步的，也就是说$T(n)$是n乘上a，再加上某个数量的b。不过b的系数要比n增长得更快。b的系数与n之间的关系可以归纳为如下：

n的值	2	4	8	16
b的系数	2	8	24	64
比率	1	2	3	4

比率是用系数b除以n的值得到的。因此，看起来b的系数是n乘上n每次翻倍便会增长1的另一个因子。具体来讲，我们可以看出这个比率是$\log_2 n$，因为$\log_2 2=1$，$\log_2 4=2$，$\log_2 8=3$，且$\log_2 16=4$。因此推测递推关系的解为$T(n)=an+bn\log_2 n$是合理的，至少对表示2的乘方的n来说如此。我们将看到该公式是正确的。

要为该递推关系求解，先遵从前面的示例中使用过的策略。我们将归纳规则写成参数m的函数，形如

对m为2的乘方且$m>1$，
$$T(m)=2T(m/2)+bm \tag{3.5}$$

接着可以从$T(n)$开始，利用(3.5)，用具有较小参数的表达式来代替$T(n)$，在这种情况下，要替换的表达式是关于$T(n/2)$的。也就是，首先有

$$T(n)=2T(n/2)+bn \tag{3.6}$$

接下来，利用(3.5)，将m替换为$n/2$，从而得到替换(3.6)中$T(n/2)$的表达式。也就是，(3.5)说明有$T(n/2)=2T(n/4)+bn/2$，而我们可以将(3.6)替换为

$$T(n)=2(2T(n/4)+bn/2)+bn=4T(n/4)+2bn$$

然后，可以用$n/4$代替(3.5)中的m，从而将$T(n/4)$替换为$2T(n/8)+bn/4$，从而得到

$$T(n)=4\big(2T(n/8)+bn/4\big)+2bn=8T(n/8)+3bn$$

我们要通过对i的归纳证明的命题就是

命题$S(i)$。如果$1\leqslant n\leqslant \log_2 n$，那么$T(n)=2^iT(n/2^i)+ibn$。

依据。对$i=1$，命题$S(1)$就是说$T(n)=2T(n/2)+bn$。这个等式是对归并排序运行时间$T(n)$的定义中的归纳规则，因此可知依据是成立的。

归纳。就像那些归纳假设是“如果……那么……”形式的归纳证明一样，如果i在假设范围之外，那么归纳步骤必定成立，这里，$i\geqslant \log_2 n$时就是这种简单的情况，这时$S(i+1)$显然成立。

再来看看困难的情况，假设$i<\log_2 n$。还要假定归纳假设$S(i)$成立，就是$T(n)=2^i T(n/2^i)+ibn$。用$n/2^i$替换(3.5)中的m，就得到

$$T(n/2^i)=2T(n/2^{i+1})+bn/2^i \tag{3.7}$$

用(3.7)的右边替换$S(i)$中的$T(n/2^i)$，就得到

$$\begin{aligned}T(n)&=2^i\left(2T(n/2^{i+1})+bn/2^i\right)+ibn\\&=2^{i+1}T(n/2^{i+1})+bn+ibn\\&=2^{i+1}T(n/2^{i+1})+(i+1)bn\end{aligned}$$

最终的等式就是命题$S(i+1)$，这样就证明了归纳步骤。

于是可以得出$S(i)$，也就是$T(n)=2^i T(n/2^i)+ibn$对在1和$\log_2 n$之间的任意i都是成立的。现在要考虑公式$S(\log_2 n)$，也就是

$$T(n)=2^{\log_2 n}T(n/2^{\log_2 n})+(\log_2 n)bn$$

我们知道$2^{\log_2 n}=n$（请回想一下，$\log_2 n$的定义就是，使2变为n，要对2乘方的次数）。还有$n/2^{\log_2 n}=1$。因此$S(\log_2 n)$可以写为

$$T(n)=nT(1)+bn\log_2 n$$

由T的定义中的依据，还知道$T(1)=a$。因此，

$$T(n)=an+bn\log_2 n$$

在经过这段分析后，必须将常数a和b替换为大O表达式，即$T(n)$是$O(n)+O(n\log n)$。[1]因为n比$n\log n$增长得更慢，所以可以忽略$O(n)$这项，直接说$T(n)$是$O(n\log n)$。也就是说，归并排序算法的时间量级是$O(n\log n)$。请记住，我们已经证明了选择排序的运行时间是$O(n^2)$。虽然严格地讲，这里的$O(n^2)$只是个上界，但是它其实是选择排序的最紧简单边界。因此，可以确定，随着n不断变大，归并排序要一直比选择排序运行得更快。从实践上讲，对值大于几十的n来说，归并排序要比选择排序更快。

3.10.4 习题

(1) 绘出下列函数的结构树。
  (a) `split`
  (b) `MergeSort`

(2) * 将k路归并排序函数定义为把一个表分为k部分，在为每部分排序后合并各部分得到结果。
  (a) 用k和n的函数表示的k路归并的运行时间是怎样的？
  (b) **什么样的k值可以带来最快的算法（用n的函数表示）？这个问题要求大家对运行时间作出足够精确的估算，从而保证自己可以区分一些常数因子。出于我们在本章开头所讨论过的原因，在实践中不可能那样精确，所以大家需要研究一下由习题(a)中得到的运行时间是怎样随着k变化的，并据此得出近似的最小值。

3.11 为递推关系求解

求解递推关系的技巧有很多。本节将讨论两种方法。第一，就是我们已经看到的，反复将

① 请记住，在大O表达式中，我们不必为对数指定底数，因为这里的对数是在常数因子中，所以所有底数的对数都是一样的。

递归规则代换到它们自身中，直到得出$T(n)$与$T(1)$的关系，或者$T(n)$与依据给出的某个$T(i)$之间的关系（如果1不是依据的话）。第二种方法是猜测一种解，并将其替换到依据和归纳规则中以验证其正确性。

在3.9和3.10两节中，我们已经为$T(n)$准确求解了。不过，因为$T(n)$实际上是确切运行时间的大O上界，所以找出$T(n)$的紧上界就够了。因此，特别是对于"猜测并验证"的方法，只需要求出的解是递推关系真正解的上界就可以了。

3.11.1　通过反复代换为递推关系求解

示例3.24所示的递推关系可能是我们在实践中遇到的最简单的递推关系了。

依据。$T(1)=a$。

归纳。对n>1，$T(n)=T(n-1)+b$。

如果可以在归纳中将常数b换成某个函数$g(n)$，就可以将这种形式进一步一般化，于是我们可以将这种形式写成下面这样。

依据。$T(1)=a$。

归纳。对n>1，$T(n)=T(n-1)+g(n)$。

只要递归函数花了时间$g(n)$，并接着用比当前函数调用所使用的参数小1的参数调用自身，就出现了这种形式。例子有示例3.24中的阶乘函数、3.10节中的`merge`函数，以及2.7节中的递归选择排序。在前两个函数中，$g(n)$是常数，而在第三个函数中，$g(n)$是n的线性函数。3.10节中的`split`函数也基本是这种形式，只不过它递归地调用自身所使用的参数是依次减小2的。我们应该明白，这种差别是不重要的。

接下来通过反复代换来求解该递推关系。正如示例3.24中那样，首先将归纳规则用参数m的函数表示出来，即

$$T(m)=T(m-1)+g(m)$$

接着反复替换原归纳规则右边的T。这样做，就可以得到一串表达式：

$$\begin{aligned}T(n)&=T(n-1)+g(n)\\&=T(n-2)+g(n-1)+g(n)\\&=T(n-3)+g(n-2)+g(n-1)+g(n)\\&\cdots\\&=T(n-i)+g(n-i+1)+g(n-i+2)+\cdots+g(n-1)+g(n)\end{aligned}$$

运用示例3.24中介绍的技巧，就可以通过对i的归纳，证明对$i=1、2、\cdots、n-1$，有

$$T(n)=T(n-i)+\sum_{j=0}^{i-1}g(n-j)$$

我们希望选择一个i值，让依据情况可以涵盖$T(n-i)$，因此我们选择$i=n-1$。因为$T(1)=a$，所以有$T(n)=a\sum_{j=0}^{n-2}g(n-j)$。换句话说，$T(n)$就是常数$a$加上从2到$n$的所有$g$之和，或者说是$a+g(2)+g(3)+\cdots+g(n)$。除非所有的$g(j)$都为0，否则在将该表达式转换为大O表达式时，$a$这项都是无关轻重的，因此一般只需要$g(j)$的和就行了。

✦ 示例 3.25

考虑一下图2-22所示的递归选择排序函数，我们在图3-29中重新展示了该函数的函数体。在需要为含m个元素的数组排序时，也就是当参数i的值为$n-m$时，如果设`SelectionSort`函

数的运行时间为$T(m)$，那么就可以得出关于$T(m)$的如下递推关系。首先，依据是$m=1$。这时，只有第(1)行执行，花的时间为$O(1)$。

```
(1)     if (i < n-1) {
(2)             small = i;
(3)             for (j = i+1; j < n; j++)
(4)                 if (A[j] < A[small])
(5)                     small = j;
(6)             temp = A[small];
(7)             A[small] = A[i];
(8)             A[i] = temp;
(9)             recSS(A, i+1, n);
            }
        }
```

图3-29 递归的选择排序

对$m>1$时的归纳，我们会执行第(1)行的测试以及第(2)、(6)、(7)、(8)行的赋值，这些语句的运行时间是$O(1)$。而第(3)至第(5)行的for循环的运行时间为$O(n-i)$，或者$O(m)$，就像我们在示例3.17中讨论过的迭代选择排序程序那样。要知道原因，请注意第(4)行和第(5)行的循环体所花的时间为$O(1)$，而我们要进行$m-1$次循环。所以，该for循环的运行时间主导了第(1)至第(8)行的运行时间，这样就可以将整个函数的运行时间$T(m)$写为$T(m-1)+O(m)$。第2项$O(m)$覆盖了第(1)至第(8)行，而$T(m-1)$这项则是第(9)行的递归调用的时间。如果将隐藏在大O表达式背后的常数因子替换为某个具体的常数，就可以得到以下递推关系。

依据。$T(1)=a$。

归纳。对$m>1$，$T(m)=T(m-1)+bm$。

该递推关系具有我们研究过的形式，其中$g(m)=b(m)$。也就是说，该递推关系的解为

$$\begin{aligned} T(m) &= a+\sum_{j=0}^{m-2} b(m-j) \\ &= a+2b+3b+\cdots+mb \\ &= a+b(m-1)(m+2)/2 \end{aligned}$$

因此$T(m)$是$O(m^2)$。我们感兴趣的是SelectionSort函数处理长度为n的整个数组时的运行时间，也就是说，当用$i=1$调用函数时，我们需要$T(n)$的表达式，并得出它是$O(n^2)$。因此，递归的选择排序是二次的，就像迭代的选择排序那样。

递推的另一种常见形式是在3.10节中为MergeSort函数得出的递推关系。

依据。$T(1)=a$。

归纳。$T(n)=2T(n/2)+g(n)$，其中n是2的乘方而且大于1。

该递推关系表示的是一个递归算法，它通过将大小为n的问题细分为两个大小为$n/2$的子问题来解决问题。这里$g(n)$是创建子问题以及结合解决方案所花的时间。例如，MergeSort将大小为n的问题分为大小为$n/2$的两个部分。函数$g(n)$具有bn的形式，其中b是某个常数，因为MergeSort除了递归调用自身之外，所花的时间是$O(n)$，主要就是用在split和merge算法上。

要求解该递推关系，需要替换等式右边的T。这里我们假设对某个k有$n=2^k$。递推关系可以写为参数为m的函数：$T(m)=2t(m/2)+g(m)$。如果用$n/2^i$替换m，就得到

$$T(n/2^i)=2T(n/2^{i+1})+g(n/2^i) \tag{3.8}$$

如果由归纳规则开始，接着用i值逐渐变大的(3.8)替换T，就会发现

$$\begin{aligned} T(n) &= 2\,T(n/2)+g(n) \\ &= 2\Big(2T(n/2^2)+g(n/2)\Big)+g(n) \\ &= 2^2T(n/2^2)+2g(n/2)+g(n) \\ &= 2^2\Big(2T(n/2^3)+g(n/2^2)\Big)+2g(n/2)+g(n) \\ &= 2^3T(n/2^3)+2^2g(n/2^2)+2g(n/2)+g(n) \\ &\cdots \\ &= 2^iT(n/2^i)+\sum_{j=0}^{i-1}2^jg(n/2^j) \end{aligned}$$

如果$n=2^k$，我们知道$T(n/2^k)=T(1)=a$。因此，当$i=k$时，也就是当$i=\log_2 n$时，可以得到递推关系的解为

$$T(n)=an+\sum_{j=0}^{(\log_2 n)-1}2^jg(n/2^j) \tag{3.9}$$

直观地讲，(3.9)的第一项表示依据值a带来的时间，也就是以大小为1的参数调用该递归函数n次的时间。而和项则是递归所花的时间，它表示以大小大于1的参数执行的所有调用的总时间。

图3-30展示了`MergeSort`函数执行期间的时间积累情况。它表示为8个元素排序的时间。第一行表示最外层的调用，涉及全部8个元素；第二行表示对两组4个元素的两次调用；第三行表示对4组两个元素的4次调用。最后，底部那行表示对长度为1的表调用`MergeSort`共8次。一般来说，如果原始无序表中有n个元素，那么通过引发其他调用的`MergeSort`调用完成bn的工作就需要$\log_2 n$层调用，因此这些调用累计的时间就是$bn\log_2 n$。还将有一层的调用不会引起进一步的调用，这些调用所花的总时间是an。请注意，前$\log_2 n$层调用表示的是(3.9)中的和项，而最下面的那层表示an那项。

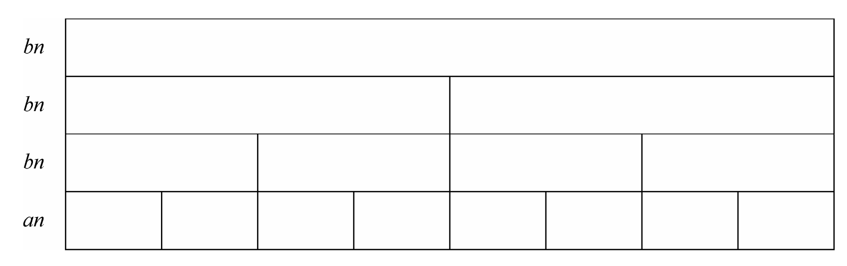


图3-30 对`MergeSort`的调用所花的时间

✦ 示例 3.26

在`MergeSort`的情况中，函数$g(n)$是bn，其中b是某个常数。因此含这些参数的(3.9)的解就是

$$\begin{aligned} T(n) &= an+\sum_{j=0}^{(\log_2 n)-1}2^jbn/2^j \\ &= an+bn\sum_{j=0}^{(\log_2 n)-1}1 \\ &= an+bn\log n \end{aligned}$$

最后得出的等式是因为和项中有 $\log_2 n$ 个项，而这些项都是1。因此，当 $g(n)$ 是线性函数时，式(3.9)的解就是 $O(n\log n)$ 。

3.11.2 通过猜测解为递推关系求解

求解递推关系的另一种实用方法就是猜测一个解 $f(n)$ ，接着使用递推关系证明 $T(n)\leqslant f(n)$ 。这可能不会给出 $T(n)$ 的精确值，不过如果它给出了紧上界，也是能令人满意的。通常我们只会猜测 $f(n)$ 这样的函数形式，不去指定一些参数。例如，我们可以猜测对某a和b，$f(n)=an^b$ 。这些参数的值都是确定的，因为我们要为所有n证明 $T(n)\leqslant f(n)$ 。

虽然可能觉得能准确猜测解是件离奇的事，但我们经常能通过观察一些较小n值所对应的 $T(n)$ 来推断出高阶项。然后就可以舍弃某些低阶项，并看看它们的系数是否非0。[①]

✦ 示例 3.27

我们再来研究一下3.10节中介绍过的MergeSort递推关系，将其写为

依据。$T(1)=a$ 。

归纳。$T(n)=2T(n/2)+g(n)$ ，其中n是2的乘方而且大于1。

我们要猜测 $T(n)$ 的上界是 $f(n)=cn\log_2 n+d$ ，其中c和d是某些常数。回想一下，这种形式并不完全正确，在之前的示例中，我们得出的解都具有 $O(n\log n)$ 项以及 $O(n)$ 项，而不带常数项。不过，这个猜测对证明 $O(n\log n)$ 是 $T(n)$ 的上界来说已经足够好了。

接着要对n进行完全归纳，证明以下命题，其中c和d是某些常数。

命题 $S(n)$ 。如果n是2的乘方而且 $n\geqslant 1$ ，那么 $T(n)\leqslant f(n)$ ，其中 $f(n)$ 是函数 $cn\log_2 n+d$ 。

依据。当 $n=1$ 时，$T(1)\leqslant f(1)$ 表示 $a\leqslant d$ ，因为当 $n=1$ 时，$f(n)$ 中 $cn\log_2 n$ 这项的值为0，则 $f(1)=d$ ，而且之前已经给定了 $T(1)=a$ 。

归纳。对所有的 $i<n$ ，假设 $S(i)$ 为真，并证明对某些 $n>1$ ，$S(n)$ 为真。如果n不是2的乘方，就没什么好证明的，因为这时具有“如果……那么……”形式的命题 $S(n)$ 的如果部分不为真。因此，考虑困难的情况，也就是n是2的乘方的情况。可以假设$S(n/2)$为真，也就是假设

$$T(n/2)\leqslant (cn/2)\log_2(n/2)+d$$

因为它是归纳假设的一部分。对归纳步骤来说，需要证明

$$T(n)\leqslant f(n)=cn\log_2 n+d$$

当 $n\geqslant 2$ 时，$T(n)$ 定义的归纳部分告诉我们

$$T(n)\leqslant 2T(n/2)+bn$$

将归纳假设应用到$T(n/2)$的边界，就有

$$T(n)\leqslant 2[c(n/2)\log_2(n/2)+d]+bn$$

因为 $\log_2(n/2)=\log_2 n-\log_2 2=\log_2 n-1$ ，所以可以将这一表达式简化为

$$T(n)\leqslant cn\log_2 n+(b-c)n+2d \tag{3.10}$$

现在要证明 $T(n)\leqslant cn\log_2 n+d$ ，条件是(3.10)右边的式子中，$cn\log_2 n+d$ 之外的部分最多为0，也就是说 $(b-c)n+d\leqslant 0$ 。因为 $n>1$ ，所以当 $d\geqslant 0$ 且 $b-c\leqslant -d$ 时该不等式成立。

要让 $f(n)=cn\log_2 n+d$ 成为 $T(n)$ 的上界，需要满足以下3条约束。

---

① 要知道，与递推关系理论很类似的微分方程理论也依赖于一些常见形式的方程的已知解，并据此通过合理的猜测求解其他方程。

(1) 约束 $a \leqslant d$ 来自依据部分。

(2) $d>0$ 来自归纳部分，不过因为已知 $a>0$，所以由(1)便可得到该不等式。

(3) $b-c \leqslant -d$，或者说 $c \geqslant b+d$，也是来自归纳部分。

如果令 $d=a$ 而且 $c=a+b$，那么这些约束显然可得到满足。我们现在就通过对n的归纳证明了，对所有大于等于1且为2的乘方的n，有

$$T(n) \leqslant (a+b)n\log_2 n + a$$

该参数表明了 $T(n)$ 是 $O(n\log n)$，也就是说 $T(n)$ 的增长速度不会比 $n\log n$ 快。不过，我们得到的边界 $(a+b)n\log_2 n + a$ 要比示例3.26中得到的确切解答 $bn\log_2 n + an$ 稍大一些，但至少是成功得到了边界。假使我们选择了更简单的猜测 $f(n)=cn\log_2 n$，可能就失败了，因为不存在可以使 $f(1) \geqslant a$ 的c值。原因在于，$c \times 1 \times \log_2 1 = 0$，这样就有 $f(1)=0$。如果 $a>0$，就显然不能使 $f(1) \geqslant a$。

不等式的处理

示例3.27中的不等式 $T(n) \leqslant cn\log_2 n + d$，是从另一个不等式 $T(n) \leqslant cn\log_2 n + (b-c)n + 2d$ 得出的。方法是找出“多余的量”，并要求它至多为0。一般的原则是，假设有不等式 $A \leqslant B+E$，那么如果要证明 $A \leqslant B$，只需要证明 $E \leqslant 0$ 就够了。在示例.27中，A是 $T(n)$，B是 $cn\log_2 n + d$，而那个“多余的量”是 $(b-c)n+d$。

✦ 示例 3.28

现在考虑的是在本书后续内容中将要遇到的一个递推关系。

依据。$G(1)=3$。

归纳。对 $n>1$，$G(n) = (2^{n/2}+1)G(n/2)$。

该递推关系包含的是实际的数字，而不是像a这样的符号常数。在第13章中，我们将使用这样的递推关系计算电路中门的数量，而且门的数量是可以准确计出的，不需要用大O表示法去隐藏不可知的常数因子。

如果我们考虑一下通过反复代换得到的解，就可能发现，要将 $G(n)$ 用含 $G(1)$ 的项表示出来，需要进行 $\log_2(n-1)$ 次代换。随着代换的不断进行，就会得到因子

$$(2^{n/2}+1)(2^{n/4}+1)(2^{n/8}+1)\cdots(2^1+1)$$

如果舍去每个因子中的“+1”项，就会近似地得出积$2^{n/2}2^{n/4}2^{n/8}\cdots 2^1$，也就是

$$2^{n/2+n/4+n/8+\cdots+1}$$

或者如果为指数部分的几何级数求和，就是 2^{n-1}，也就是 2^n 的一半，因此可以猜测，2^n 是解 $G(n)$ 中的项。不过，如果猜测 $f(n)=c2^n$ 是 $G(n)$ 的上界，就可能会求解失败，读者可以自行验证。也就是说，我们得到了两个涉及c的不等式，但它们不是解。

因此我们会猜测下一种最简单的形式，$f(n)=c2^n+d$，而这样就能成功求解了。也就是说，可以通过对n的完全归纳证明以下命题，其中c和d是某些常数。

命题 $S(n)$。如果n是2的乘方且 $n \geqslant 1$，那么 $G(n) \leqslant c2^n + d$。

依据。如果 $n=1$，那么必须证明 $G(1) \leqslant c2^1 + d$，也就是 $3 \leqslant 2c+d$。该不等式变成了对c和d的约束。

归纳。像示例3.27那样，唯一的难点出现在当n为2的乘方而且要从$S(n/2)$证明$S(n)$时。这种情况下等式就成了

$$G(n/2) \leqslant c2^{n/2} + d$$

必须证明$S(n)$，也就是$G(n) \leqslant c2^n + d$。首先从G的归纳定义开始。

$$G(n) = (2^{n/2} + 1)G(n/2)$$

然后用得到的上界替换$G(n/2)$，就将上述表达式转换成了

$$G(n) \leqslant (2^{n/2} + 1)(c2^{n/2} + d)$$

加以简化，就得到

$$G(n) \leqslant c2^n + (c+d)2^{n/2} + d$$

这要给出所需的$G(n)$的上界$c2^n + d$，因此就要有右侧多余的部分$(c+d)2^{n/2}$不大于0，而这只需有$c + d \leqslant 0$就足够了。

我们需要选择c和d来满足两个不等式。

(1) 源于依据部分的$2c + d \geqslant 3$。

(2) 源于归纳部分的$c + d \leqslant 0$。

例如，如果$c = 3$且$d = -3$，则两个不等式都能满足。那么我们就知道$G(n) \leqslant 3(2^n - 1)$。因此，$G(n)$是随着$n$指数增长的。刚好这个函数就是确切的解，也就是说$G(n) = 3(2^n - 1)$，读者可以自己通过对$n$的归纳来证明这一点。

对解的总结

下面的表格中列出了一些最常见的递推关系，其中包括本节未曾介绍的。在每种情况中，假设依据等式为$T(1) = a$，而且有$k>0$。

归纳等式	$T(n)$
$T(n) = T(n-1) + bn^k$	$O(n^{k+1})$
$T(n) = cT(n-1) + bn^k$ 其中 $c>1$	$O(c^n)$
$T(n) = cT(n/d) + bn^k$ 其中 $c>d^k$	$O(n^{\log_d c})$
$T(n) = cT(n/d) + bn^k$ 其中 $c<d^k$	$O(n^k)$
$T(n) = cT(n/d) + bn^k$ 其中 $c = d^k$	$O(n^k \log n)$

若上述等式中的bn^k被替换为任意的k次多项式，其结论也都是成立的。

3.11.3 习题

(1) 设$T(n)$是由如下递推关系定义的

$$对\ n>1,\ T(n) = T(n-1) + g(n)$$

通过对i的归纳证明，如果$1 \leqslant i \leqslant n$，那么

$$T(n) = T(n-i) + \sum_{j=0}^{i-1} g(n-j)$$

(2) 假设有如下形式的递推关系

$T(1)=a$

对 $n>1$， $T(n)=T(n-1)+g(n)$

如果 $g(n)$ 分别是

(a) n^2

(b) n^2+3n

(c) $n^{3/2}$

(d) $nlogn$

(e) 2^n

给出解的紧大O上界。

(3) 假设有如下形式的递推关系

$T(1)=a$

当n是2的乘方且 $n>1$ 时， $T(n)=T(n/2)+g(n)$

如果 $g(n)$ 分别是

(a) n^2

(b) $2n$

(c) 10

(d) $nlogn$

(e) 2^n

给出解的紧大O上界。

(4) *将下列各项作为如下递推关系的解进行猜测。

$T(1)=a$

当n是2的乘方且 $n>1$ 时， $T(n)=2T(n/2)+bn$

(a) $cn\log_2 n+dn+e$

(b) $cn+d$

(c) cn^2

这暗示了未知常数c、d和e所具有的哪些约束？对哪些形式而言，存在 $T(n)$ 的上界？

(5) 证明：如果我们为示例3.28中的递推关系猜测 $G(n)\leqslant c2^n$ ，那么将没法找到解。

(6) * 证明：如果

$T(1)=a$

对 $n>1$， $T(n)=T(n-1)+n^k$

那么 $T(n)$ 是 $O(n^k)$ 。大家可以假设 $k\geqslant 0$ 。证明这是 $T(n)$ 的最紧简单大O上界，也就是说，如果 $m<k+1$ ， $T(n)$ 就不是 $O(n^m)$ 了。提示：用 $T(n-i)$ （其中 $i=1$、2、… ）展开 $T(n)$ ，从而得到上界。要得到下界，就要证明对某个特定的 $c>0$ 而言， $T(n)$ 至少是 cn^{k+1} 。

(7) ** 证明：如果

$T(1)=a$

对 $n>1$， $T(n)=cT(n-1)+P(n)$

其中 $p(n)$ 是n的任意多项式且 $c>1$ ，那么 $T(n)$ 是 $O(c^n)$ 。还有，证明这是最紧简单大O上界，也就是说，如果 $d<c$ ，那么 $T(n)$ 不是 $O(d^m)$ 。

(8) ** 考虑递推关系

$T(1)=a$

当n是d的乘方时， $T(n)=cT(n/d)+bn^k$

用 $T(n/d^i)$ （其中 $i=1$、2、… ）迭代地展开 $T(n)$ ，证明

(a) 如果 $c>d^k$ ，那么 $T(n)$ 是 $O(n^{\log_d c})$

(b) 如果 $c=d^k$ ，那么 $T(n)$ 是 $O(n^k\log n)$

(c) 如果 $c < d^k$ ，那么 $T(n)$ 是 $O(n^k)$

(9) 求解以下递推关系，其中每个关系都有 $T(1) = a$ 。

(a) 当n是2的乘方且 $n>1$ 时， $T(n) = 3T(n/2) + n^2$

(b) 当n是3的乘方且 $n>1$ 时， $T(n) = 10T(n/3) + n^2$

(c) 当n是4的乘方且 $n>1$ 时， $T(n) = 16T(n/4) + n^2$

可能会用到习题(8)中的解答。

(10) 求解递推关系

$T(1) = a$

当n是2的乘方且 $n>1$ 时， $T(n) = 3^n T(n/2)$

(11) 斐波那契递推关系是 $F(0) = F(1) = 1$ ，且

对 $n>1$ ， $F(n) = F(n-1) + F(n-2)$

来自该数列的值 $F(0)$ 、$F(1)$ 、$F(2)\cdots$就是斐波那契数，其中从第3个数起，每个数都是相邻前两个数的和，见3.9节习题(4)。设 $r = (1+\sqrt{5})/2$ ，该常数r称为*黄金比例*，而且其值约为1.62。证明：$F(n)$ 是 $O(r^n)$ 。*提示*：对于归纳部分，可以猜测对某个n有 $F(n) \leqslant ar^n$ ，并尝试通过对n的归纳证明该不等式。依据必须是由 $n=0$ 和 $n=1$ 这两个值组成。在归纳步骤中，可以注意到r满足 $r^2 = r+1$ 这一关系。

3.12　小结

以下是本章涵盖的一些重点概念。

- 许多因素会对程序算法的选择产生影响，不过通常简单、易于实现和高效起主导作用。
- 大O表达式提供了一种很方便的程序运行时间上界表示法。
- 在评估C语言复合语句（比如`for`循环和条件语句）的运行时间时，存在一些递归规则，这些规则是用这些复合语句各组成部分的运行时间表示的。
- 通过绘制表示语句嵌套结构的结构树，并按照从下至上的顺序评估结构树中各部分的运行时间，可以评估函数的运行时间。
- 递推关系是为递归程序运行时间建模的一种自然方法。
- 要为递推关系求解，既可以通过反复代换，也可以通过先猜测解并验证猜测为正确的方式。

分治法是一种重要的算法设计技巧。使用分治法时会将问题分为若干个子问题，这些子问题的解答将会组合成整个问题的解答。可根据一些经验法则来评估由此产生的算法（运行时间为 $O(1)$ ，且对大小为 $n-1$ 的子问题调用自身所花时间为 $O(n)$ ）的运行时间。这种算法的例子包括阶乘函数和`merge`函数。

- 更为一般的情况是，函数花的时间为 $O(n^k)$ ，而且对大小为 $n-1$ 的子问题调用自身花的时间是 $O(n^{k+1})$ 。
- 如果函数调用自身两次，而递归行进了 $\log_2 n$ 层（就像归并排序那样），那么总运行时间就是 $O(n\log n)$ 乘上每次调用的开销，再加上 $O(n)$ 乘上依据部分的开销。在归并排序中，包括依据调用在内，每次调用的开销都是$O(1)$，所以总运行时间就是 $O(n\log n) + O(n)$ ，或者是 $O(n\log n)$ 。
- 如果函数调用自身两次，而递归行进了n层，就像3.9节习题(4)的斐波那契程序那样，那么运行时间就是指数为n的指数形式。

3.13　参考文献

对程序运行时间及问题计算复杂度的研究始于Hartmanis and Stearns [1964]。Knuth [1968]这本书使算法运行时间的研究成为了计算机科学的重要部分。

自那时起，大量有关问题难度的理论涌现出来。其中很多关键的概念都能在Aho, Hopcroft, and Ullman [1974, 1983]中找到。在本章中，我们将精力都集中在了程序运行时间的上界上。而Knuth [1976]则描述了类似的下界表示法以及运行时间的精确边界。

要了解更多有关分治法这种算法设计技巧的讨论，请参考Aho, Hopcroft, and Ullman [1974]或Borodin and Munro [1975]。更多与求解递推关系的技巧有关的信息可以在Graham, Knuth, and Patashnik [1989]中找到。

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1974]. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1983]. *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass.

Borodin, A. B., and I. Munro [1975]. *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York.

Graham, R. L., D. E. Knuth, and O. Patashnik [1989]. *Concrete Mathematics: a Foundation for Computer Science*, Addison-Wesley, Reading, Mass.

Knuth, D. E. [1968]. *The Art of Computer Programming* Vol. I: *Fundamental Algorithms,* Addison-Wesley, Reading, Mass.

Knuth, D. E. [1976]. “Big omicron, big omega, and big theta,” *ACM SIGACT News* 8:2, pp. 18–23.

第 4 章

组合与概率

在计算机科学中，我们常需要为事物计数，并度量事件的可能性。计数属于数学中的组合学分支，而度量事件的可能性则属于概率论的范畴。本章要介绍这两个领域的基本原理。我们会了解到一些问题的答案，诸如程序中有多少条执行通路（execution path），或给定通路出现的可能性有多大等。

4.1 本章主要内容

本章给出了一系列越来越复杂的情况，每种情况都通过一个简单的范例问题说明，借此对组合（或“计数”）加以研究，而且会为每个问题推导出用于确定可能结果数量的公式。要研究的问题包括以下几个。

- 为分配计数（4.2节）。范例问题是：用k种颜色为n所房屋粉刷共有多少种不同方式。
- 为排列计数（4.3节）。范例问题是：确定n个不同项能构成多少种不同的次序。
- 为有序选择计数（4.4节），也就是从n个不同事物中选出k个，并按次序排列这k个事物。范例问题是：计算赛马比赛中不同马匹获得前三名的排列方法数。
- 为n个事物中的m个的组合计数（4.5节），也就是从n个不同对象中选择m个，而不考虑被选取对象的次序。范例问题是：为可能的扑克牌型计数。
- 为具有某些重复项的排列计数（4.6节）。范例问题是：计算某些字母多次出现的单词的变位词的数量。
- 为分发容器中对象（可能具有重复对象）的方法计数（4.7节）。范例问题是：为给小朋友分发水果的方法计数。

本章的后半部分要讨论的是概率论，涵盖以下主题。

- 基本概念：概率空间、实验、事件、事件概率。
- 条件概率与事件独立性。这些概念可帮助我们了解，对一次实验结果（比如纸牌的牌面图案）的观察会怎样影响未来事件的概率。
- 概率推理和方法。通过这些推理和方法，可从与事件的概率及条件概率相关的有限数据中，估算出事件组合的概率。

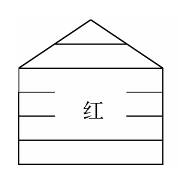
我们还将讨论概率论在计算机领域的一些应用，包括根据数据进行或然性推理的系统，以及一类“有很大概率”有效但不保证一直有效的算法。

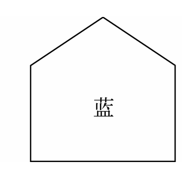
4.2　为分配计数

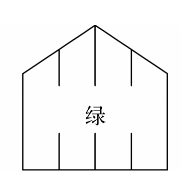
一种简单却极重要的计数问题是处理一组项，为每一项指定某一组固定值中的某个值。我们需要确定可能有多少种将值分配给项的方式。

✦　示例 4.1

图4-1展示了一个典型的例子，其中有并排4所房屋，而且可将每所房屋粉刷成红、绿、蓝这三种颜色中的一种。在本例中，房屋就是之前所提到的“项”，而颜色就是“值”。图4-1展示了一种可能的颜色分配方式，其中第一所房屋被刷成红色，第二所和第四所被刷成蓝色，而第三所被刷成绿色。







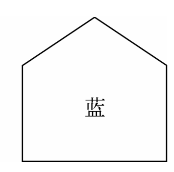


图4-1　房屋颜色分配的一种方式

要回答“有多少种不同分配方式”，首先需要定义我们所说的“分配”具有何种含义。在本例中，一种分配方式就是一个具有4个值的表，其中每个值都是从红、绿、蓝这3种颜色中任选其一。接下来要分别用字母R、G和B来表示这3种颜色。而当且仅当两个这样的表至少有一个位置不同时，我们称这两个表是*不同的*。

在这个房屋与颜色的例子中，可以为第一所房屋任选三种颜色之一。不管为第一所房屋选择了什么颜色，在粉刷第二所房屋时还是有这三种选择。因此粉刷前两所房屋的方式有9种，对应着9个不同的字母对，每个字母都是R、G和B这三者之一。类似地，对前两所房屋所具有的9种分配方式的每一种而言，都可以为第三所房屋在三种颜色中任选其一。这样一来，前三所房屋的粉刷方式就达到了$9\times3=27$种。最后，这27种分配方式中对应的第四所房屋又都能在三种颜色中任选其一，因此总共有$27\times3=81$种粉刷房屋的方式。

4.2.1　为分配计数的规则

可以对以上示例加以扩展。在一般情形下，有一列n个“项”，比如示例4.1中的房屋；还有一组k个“值”，如示例4.1中的颜色，可以给某个项指定这些值中的任一种。一种*分配*就是一个含有n个值的表$(v_1, v_2, \cdots, v_n)$。$v_1, v_2, \cdots, v_n$中的每一个都是从这k个值中任选其一。这种分配指定了v_i从v_1到第i项的值，其中$i=1, 2, \cdots, n$。

当有n个项，而且可以为每一项指定k个值之一时，就会有k^n种不同的分配。例如，在示例4.1中，一共有$n=4$项，也就是有4所房屋，而且有$k=3$个值，也就是有3种颜色。我们就可以计算出总共有81种不同的分配。请注意，就是$3^4=81$种。可以通过对n的归纳证明这一一般规则。

命题 $S(n)$。为n个项中每一项分配k个值中的任一个，共有k^n种方式。

依据。依据为$n=1$的情况。如果只有一项，可以为它任选k个值中的一个。因为$k^1=k$，所以依据得证。

归纳。假设命题$S(n)$为真，并考虑$S(n+1)$，也就是为$n+1$项分别分配k个值之一，共有k^{n+1}种方式。可以将这种分配分解为给第一项选择值，以及针对第一个值的每种选择，为剩下的n项分配值。对每种这样的选择而言，根据归纳假设，剩下的n项有k^n种分配值的方式。所以总分配方式共有$k \times k^n$种，也就是有k^{n+1}种。因此我们证明了$S(n+1)$，完成了归纳步骤。

图4-2表示了当$n+1=4$且$k=3$时，即在示例4.1这个讨论4所房屋和3种颜色的具体例子中，对第一个值的选择以及相应的剩余项分配方式的选择。也就是说，在归纳假设中假定选择3种颜色之一粉刷3所房屋共有27种分配方式。

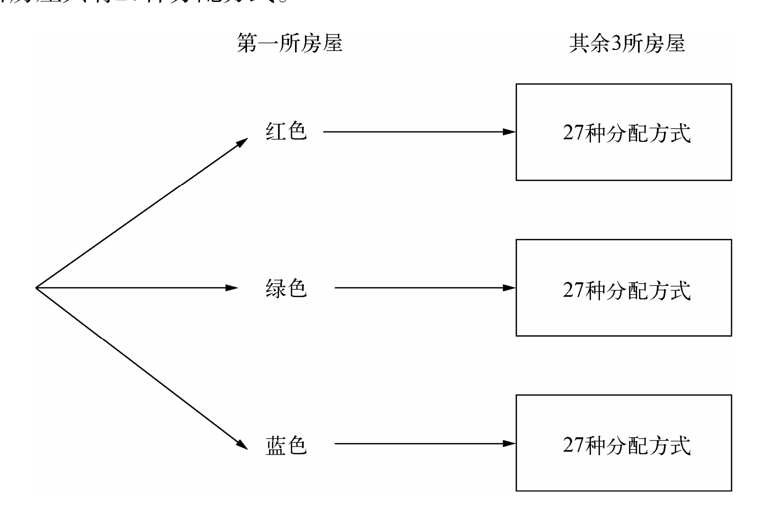


图4-2 用3种颜色粉刷4所房屋的分配方式数

4.2.2 为位串计数

在计算机系统中，我们常遇到由0和1组成的串，而这些串往往用作对象的名称。例如，我们可能购买具有“64MB主内存”的计算机。每一个字节都有自己的名称，而这个名称是长度为26位的位序列，每一位要么是0，要么是1。这种由0和1组成的表示名称的串就叫作*位串*。

为什么对64MB的内存来说是26位呢？答案就源自分配计数问题。当我们计算长度为n的位串的数量时，可以将串中的位置视作“项”，而这些位置可能存放0或1这两个值中的一个。因为有两个值，所以有$k=2$，而为n个项分配二值之一的分配方式共有2^n种。

如果$n=26$，即考虑长度为26的位串，就可能有2^{26}种位串。2^{26}的精确值为67 108 864。而按照计算机的语法，这个数字会被视为“6 400万”，虽然真实的数字显然要比这个值大上约5%。接下来的附注栏简要介绍了该主题，并试着解释了为2的乘方命名时涉及的一般规则。

K、M和2的乘方

将2的乘方转换成10的乘方有个实用的技巧。我们可以注意到，2^{10}，也就是1024，与1000是非常接近的。因此，2^{30}，也就是$(2^{10})^3$，或者说大概是1000^3，即10亿。那么，$2^{32}=4\times 2^{30}$，也就是约40亿。其实，计算机科学家通常都会认可2^{10}正好是1000的假设，并将2^{10}说成是1K，其中K表示kilo（千）。例如，我们可将2^{15}转换成32K，因为

$$2^{15}=2^5\times 2^{10}=32\times \text{“1000”}$$

而我们将实际值为1 048 576的 2^{20} 称为1M，或者是1兆，而不是称为1000K或1024K。对 2^{20} 到 2^{29} 这几个2的乘方数，我们会提取出 2^{20} 这个因子。因此，2^{26} 就是 $2^6 \times 2^{20}$，或者说是64兆。这正是 2^{26} 字节被称为64兆字节或64 MB的原因。

下表给出了多项10的乘方，以及与其近似相等的2的乘方。

前缀	字母	值
Kilo	K	10^3 或 2^{10}
Mega	M	10^6 或 2^{20}
Giga	G	10^9 或 2^{30}
Tera	T	10^{12} 或 2^{40}
Peta	P	10^{15} 或 2^{50}

本表格表明对超过 2^{29} 的2的乘方，我们分别会提取出 2^{30} 、2^{40} 或是可以达到的2的任意整十次方作为因子。不管用什么单位度量，剩下的2的乘方会在命名时加上giga-、tera-或peta-这些前缀。例如，2^{43} 字节就是8TB。

4.2.3　习题

(1) 在下列情形中，分别有多少种粉刷方式?
  (a) 3所房屋，每一所可从4种颜色中任选一种。
  (b) 5所房屋，每一所可从5种颜色中任选一种。
  (c) 2所房屋，每一所可从10种颜色中任选一种。

(2) 假设计算机密码由8到10位字母和（或）数字组成。可能有多少种不同的密码？请记住，大写字母和小写字母是不同的。

(3) * 考虑如图4-3所示的f函数。f可以返回多少种不同的值？

```
int f(int x)
{
    int n;

    n = 1;
    if (x%2 == 0) n *= 2;
    if (x%3 == 0) n *= 3;
    if (x%5 == 0) n *= 5;
    if (x%7 == 0) n *= 7;
    if (x%11 == 0) n *= 11;
    if (x%13 == 0) n *= 13;
    if (x%17 == 0) n *= 17;
    if (x%19 == 0) n *= 19;
    return n;
}
```

图4-3　f函数

(4) 在“好莱坞广场”游戏中，X和O可能以任意组合被放置在井字棋棋盘（一个3×3的矩阵）9个格子的任意一个中（即与普通井字棋玩法不同的是，这里的X和O不必要交替摆放，所以，打个比方，所有的格子都可以放上X）。方阵也可能为空，也就是说，既不含X，也没有O。那么有多少种不同的摆放方法呢?

(5) 用10个数字可以组成多少种长度为n的串？其中某个数字可能出现任意次，也可能根本不出现。

(6) 用26个小写字母可以组成多少种长度为n的串？其中某个字母可以出现任意次，也可能根本不出现。

(7) 根据上文附注栏中所述的规则，将以下内容转换成K、M、G、T或P：(a) 2^{13} (b) 2^{17} (c) 2^{24} (d) 2^{38} (e) 2^{45} (f) 2^{59} 。

(8) * 将以下10的乘方转换成近似的2的乘方：(a) 10^{12} (b) 10^{18} (c) 10^{99} 。

4.3　为排列计数

本节中我们将解决另一个基础的计数问题：将给定的n个不同对象排成一列，可以有多少种不同的排列方式？这种排序称为这些对象的*排列*。我们将用$\Pi(n)$表示n个对象的排列数。

关于为排列计数在计算机科学中的重要性，我们来举例说明。假设要为给定的n个对象a_1、a_2、…、a_n排序。如果对这些对象一无所知，那么任何次序都可能是正确的排序次序，因此排序可能的结果数就是$\Pi(n)$，也就是n个对象的排列数。我们很快就会看到，这一结果有助于证实：通用的排序算法所需的时间与$n\log n$成正比，并因此可证实3.10节中运行时间为$O(n\log n)$的归并排序算法会快上某个常数因子倍。

排列计数规则还有很多其他应用。例如，我们将在后面的小节中看到的，它在组合与概率这样更为复杂的计数问题中也分量十足。

✦　示例 4.2

为了直观，我们列举一下微量对象的排列。首先，显然有$\Pi(1)=1$。也就是说，如果只有一个对象A，就只有一种次序A。

然后考虑有两个对象A和B的情况。可以从两个对象中任选其一排列在第一位，而将另一个对象排列在第二位，因此有两种次序：AB和BA。所以$\Pi(2)=2\times1=2$。

接着看看有3个对象A、B和C的情况。可以从三者中任选其一排在首位。先考虑选择A排在第一位的情况，这时候剩下B和C这两个对象，它们可以按两个对象的两种次序之一分布，从而完成这一排列。因此可以看出，由A开头的排列有两种，即ABC和ACB。

类似地，如果以B开头，也有两种方式完成这一序列，对应为剩下的对象A和C排序的两种方式，因此有序列BAC和BCA。最后，如果以C开头，就可以用两种方式为剩下的对象A和B排序，从而得到序列CAB和CBA。ABC、ACB、BAC、BCA、CAB和CBA这6个序列就是3个元素可能排成的所有次序了。也就是说，$\Pi(3)=3\times2\times1=6$。

接下来考虑一下4个对象A、B、C和D可以形成多少排列。如果选择A排在首位，那么跟在A之后的对象B、C和D可以按照6种次序中的任意一种排列。类似地，如果将B排在第一位，那么剩下的A、C和D也能按6种次序排列。现在一般模式应该明了了。可以从4个元素中任选一个排在第一位，而对每种选择，都可以按照$\Pi(3)=6$种可能方式中的任意一种排列剩余元素。请注意，3个对象的排列数并不取决于这3个元素到底是什么。由此可以得出结论：4个对象的排列数等于4乘以3个对象的排列数。

一般而言，对任意$n\geqslant1$，有$\Pi(n+1)=(n+1)\Pi(n)$　(4.1)

也就是说，要为$n+1$个对象的排列计数，可以从$n+1$个对象中任选一个排在首位。然后剩下的n个对象可以有$\Pi(n)$种排列方式，如图4-4所示。在我们的例子中，$n+1=4$，于是有$\Pi(4)=4\times\Pi(3)=4\times6=24$。

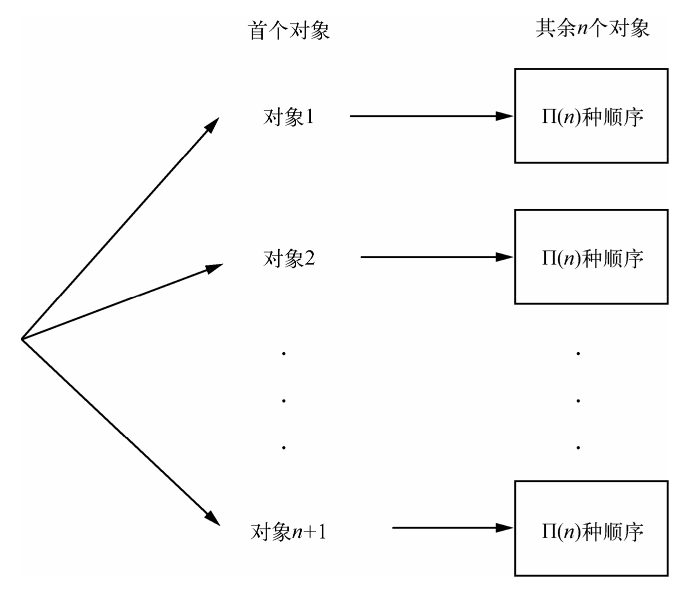


图4-4　$n+1$个对象的排列

4.3.1　排列公式

等式(4.1)就是2.5节介绍的阶乘函数定义中的归纳步骤。因此不用为$\Pi(n)$等于$n!$感到惊讶。我们可以通过简单的归纳证明这种等价性。

命题 $S(n)$。对所有的$n \geqslant 1$，有$\Pi(n)=n!$。

依据。对$n=1$，$S(1)$表示1个对象有1种排列。我们在示例4.2中已经看出这一点了。

归纳。假设$\Pi(n)=n!$。那么要证明的$S(n+1)$就是$\Pi(n+1)=(n+1)!$。由等式(4.1)，有

$$\Pi(n+1)=(n+1)\times\Pi(n)$$

而根据归纳假设，$\Pi(n)=n!$。因此，$\Pi(n+1)=(n+1)\times n!$。因为

$$n!=n\times(n-1)\times\cdots\times 1$$

所以一定有$(n+1)\times n!=(n+1)\times n\times(n-1)\times\cdots\times 1$。而后者的积就是$(n+1)!$，这就证明了$S(n+1)$为真。

✦　示例 4.3

根据公式$\Pi(n)=n!$，可以得出结论：4个对象的排列数是$4!=4\times 3\times 2\times 1=24$，正如我们在上面所见的。再举个例子，7个对象的排列数就是$7!=5040$。

4.3.2　排序要花多久

该排列计数公式有个有趣的用途，就是可用来证明，要为n个元素排序，排序算法至少会花上与$n\log n$成正比的某段时间，除非在排序过程中利用到这些元素的某些特殊属性。例如，在后文附注栏有关特例排序算法的介绍中，可以注意到，如果编写只处理较小整数的排序算法，就可以使运行时间比与$n\log n$成正比的值更少。

不过，如果某个排序算法可以处理任意种类的数据，那么只要这些数据可以通过某种“小于”关系进行比较，该算法确定合适次序的唯一方式就是考量两个元素中的一个是否小于另一

个。如果某种排序算法对待排序元素的唯一操作是比较二者以确定它们的相对次序，那么这种算法就可称为*通用排序算法*（general-purpose sorting algorithm）。例如，第2章中介绍的选择排序和归并排序都是这样作出决定的。即便编写的程序是用来处理整数数据的，也可以将其编写得更具一般性，只要将图2-2第(4)行中

```
if (A[j] < A[small])
```

这样的比较替换成诸如

```
if (lessThan(A[j], A[small]))
```

这类调用布尔值函数的测试即可。

假设有n个不同的元素有待排序。答案（也就是正确的排序次序）可能是这些元素形成的$n!$种排列中的任意一种。如果用于为任意类型的元素排序的算法能正常工作，它就一定能区分这$n!$种不同的可能答案。

考虑该算法进行的第一次元素比较，假设是

```
lessThan(X,Y)
```

对这$n!$种可能的排序次序而言，X要么小于Y，要么不小于Y。因此，这$n!$种可能的次序会被划分为两组，分别是第一次测试的答案为“是”的组，以及答案为“否”的组。

这两组中的一组必须至少具有$n!/2$个成员，因为如果两个组的成员都不足$n!/2$个，总的次序数就少于$n!/2+n!/2$个，也就是少于$n!$种次序。而这一次序数量的上限就限制了我们刚好有$n!$种次序。

现在考虑第二个测试，假设对X和Y进行比较的结果是得出如下结论：两组可能的次序中较大的那组会剩下（如果这两组一样大则任取一组）。也就是说，至少会剩余$n!/2$种次序必须由算法来区分。第二次比较同样有两种可能的结果，而且剩余的次序中至少有一半会与这些结果之一相同。因此，我们会发现，至少有$n!/4$种次序与前两次测试的结果一致。

可以重复这一论证，直到算法确定正确的排序次序为止。在每一步中，只要将重点放在含有较多一致可能次序的结果上，就至少会留下一半上一步中得到的可能次序。因此，可以看到这样一系列的测试和结果：在第i次测试后，至少有$n!/2^i$种次序与这些结果相一致。

因为直到每个测试和结果序列最多与一个排序次序一致才会完成排序，所以在完成排序前所进行测试的次数t要满足

$$n!/2^t \leqslant 1 \tag{4.2}$$

如果对(4.2)式的两边同时取以2为底的对数，就得到$\log_2 n!-t \leqslant 0$，也就是

$$t \geqslant \log_2(n!)$$

我们将看到$\log_2(n!)$大约是$n\log_2 n$。不过首先要看一个分割可能次序的示例。

✦　示例 4.4

考虑一下图2-2所示的选择排序算法在为给定的3个元素（a, b, c）排序时是如何作出判定的。第一次比较发生在a和b之间，如图4-5中的顶端所示，其中方框中表示了进行任何测试前，6种可能的次序全部是一致的。在测试后，abc、acb和cab这些次序与结果为“是”（即$a<b$）的情况一致，而bac、bca和cab这些次序与相反的结果（也就是$b>a$）一致。我们再次在方框中展示了每种情况中的一致序（consistent order）。

在图2-2所示的算法中，较小元素的下标成了变量`small`的值。因此，接下来要将c与a和b中的较小者进行比较。请注意，接下来要进行何种测试取决于上一次测试的结果。

在进行第二次判定后，3个元素中最小的那个会被移动到数组的第一个位置，而第三次比较则会确定剩下的两个元素中哪个更大。第三次比较是该算法在为3个元素排序时所要进行的最后一次比较。正如我们在图4-5的底部看到的，有时候判定的结果是确定的。例如，如果已经得到 $a<b$ 而且 $c>b$ ，那么c 就是最小的元素，而且最后一次对a和b的比较会得出a更小的结论。

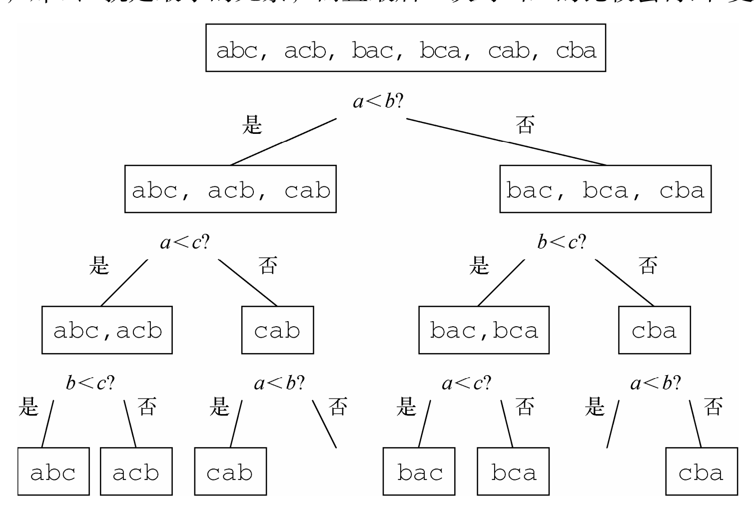


图4-5　对3个元素进行选择排序的判定树

在本示例中，所有路径都包含3次判定，而且最后至多存在一种一致序，就是正确的排序次序。不含一致序的两条路径从未出现。(4.2)式说明测试次数t一定至少为 $\log_2 3!$，即 $\log_2 6$ 。由于6大于 2^2 且小于 2^3 ，所以可知 $\log_2 6$ 大于2小于3。所以，为3个元素排序的任意算法至少有某个结果序列必须进行3次测试。因为选择排序只需为3个元素进行3次测试，所以处理3个元素时，它最不济也至少与其他算法一样好。当然，随着元素数量不断变多，选择排序就不那么好了，因为它是种 $O(n^2)$ 的排序算法，而且还存在更佳的算法，比如归并排序。

现在必须要估算 $\log_2 n!$ 有多大。因为$n!$是从1到n这n个整数的积，它肯定要比从 $n/2$ 到n这 $\frac{n}{2}+1$ 个整数的积大。这 $\frac{n}{2}+1$ 个整数的积又至少与 $n/2$ 个 $n/2$ 的积，也就是 $(n/2)^{n/2}$ 一样大。因此，$\log_2 n!$ 至少是 $\log 2\left((n/2)^{n/2}\right)$ ，即 $\frac{n}{2}(\log_2 n - \log_2 2)$ ，也就是

$$\frac{n}{2}(\log_2 n - 1)$$

对较大的n来说，这一公式约等于 $(n\log_2 n)/2$ 。

更细致的分析将表明常数因子1/2在这里并非必要。也就是说，$\log_2 n!$ 非常接近 $n\log_2 n$ ，而非更接近它的一半。

线性时间的专用排序算法

如果对排序算法可以处理的输入加以限制，就可以在一个步骤中将可能的次序分为2个以上的部分，因此会让运行时间少于与 $n\log n$ 成正比的时间。下面讲一个简单例子，如果输入是n个从0到 $2n-1$ 之间的不同整数，它就能起作用。

```
(1)  for (i = 0; i < 2*n; i++)
(2)      count[i] = 0;
(3)  for (i = 0; i < n; i++)
(4)      count[a[i]]++;
(5)  for (i = 0; i < 2*n; i++)
(6)      if (count[i] > 0)
(7)          printf("%d\n", i);
```

假设输入为长度为n的数组a。在第(1)行和第(2)行，我们将长度为$2n$的数组count初始化为0。接着，在第(3)行和第(4)行中，若x为第i个输入元素a[i]的值，则为x的计数加上1。在最后3行代码中，要打印出count[i]为正的各个整数i。因此，要打印那些在输入中至少出现过一次的元素，而之前假设了输入中各元素都是不同的，所以这段代码会将所有的输入元素按照从小到大的顺序打印出来。

分析该算法运行时间很容易。第(1)行和第(2)行是一个会迭代$2n$次的循环，而且其循环体的运行时间为$O(1)$。因此，该循环的运行时间为$O(n)$。同理，第(3)行和第(4)行的循环运行时间也是$O(n)$，只不过它的迭代次数是n。最后，第(5)行至第(7)行所示循环的循环体运行时间为$O(n)$，而它会迭代$2n$次。因此，这3个循环的运行时间均为$O(n)$，而整个排序算法的运行时间同样是$O(n)$。请注意，如果给定的输入没有为该算法进行过处理，比如输入中含有超出从0到$2n-1$范围的整数，那么上面的程序就无法正确排序。

我们只是证实了任意通用排序算法都一定有某些能让它们进行$n\log_2 n$或更多次比较的输入。因此，任意通用排序算法在最坏的情况下肯定至少要花上与$n\log n$成正比的时间。其实，可以证明，这一点同样适用于“平均的”输入。也就是说，通用排序算法处理所有输入平均所花的时间一定至少与$n\log n$成正比。因此，归并排序基本上就是我们能做的最佳算法了，因为它处理所有输入都有着这样的大O运行时间。

4.3.3　习题

(1) 假设已经为棒球队选择了9名队员。

(a) 可能存在多少种击球次序？

(b) 如果投手必须最后击球，那么可能有多少击球次序？

(2) 如果要为4个元素排序，那么图2-2中的选择排序算法要进行多少次比较？这是不是可以达到的最优数字？给出该情况下判定树（具有如图4-5所示样式）最上面的3层。

(3) 2.8节介绍的归并排序算法在处理4个元素时要进行多少次比较？这是否为可达到的最优数字？给出该情况下判定树（具有如图4-5所示样式）最上面的3层。

(4) * 将n个值分配给n个项的数目多，还是$n+1$个项的排列数多？请注意：对不同的n来说，答案可能不同。

(5) * 将$n/2$个值分配给n个项的数目是否多于n个项的排列数？

(6) ** 说明如何在$O(n)$时间内为范围在0到n^2-1之间的n个整数排序。

4.4　有序选择

有时候我们会希望只从集合中选出某些项，并为它们排定顺序。这里将4.3节中介绍过的为排列计数的函数$\Pi(n)$一般化为双参数的函数$\Pi(n,m)$，用该函数表示从n个项中选出m项排定次序的方法数，不过对未选定的项来说没有次序可言。因此$\Pi(n)=\Pi(n,n)$。

✦ 示例 4.5

赛马比赛会为前三名完成比赛的赛马颁奖。假设有10匹马参赛，那么冠亚季军的排列情况共有多少种呢？

显然，10匹马中的任意一匹都可能赢得比赛。如果给定了获得冠军的马匹，那么剩下的9匹马可以任意排序。因此前两名马匹的选择共有 $10\times9=90$ 种。对每种选择而言，都会剩下8匹赛马，其中任意一匹都可能获得季军。因此，冠亚季军的选择方式共有 $90\times8=720$ 种。图4-6展示了所有可能的选择，重点突出了首先选择3号之后选择1号的情况。

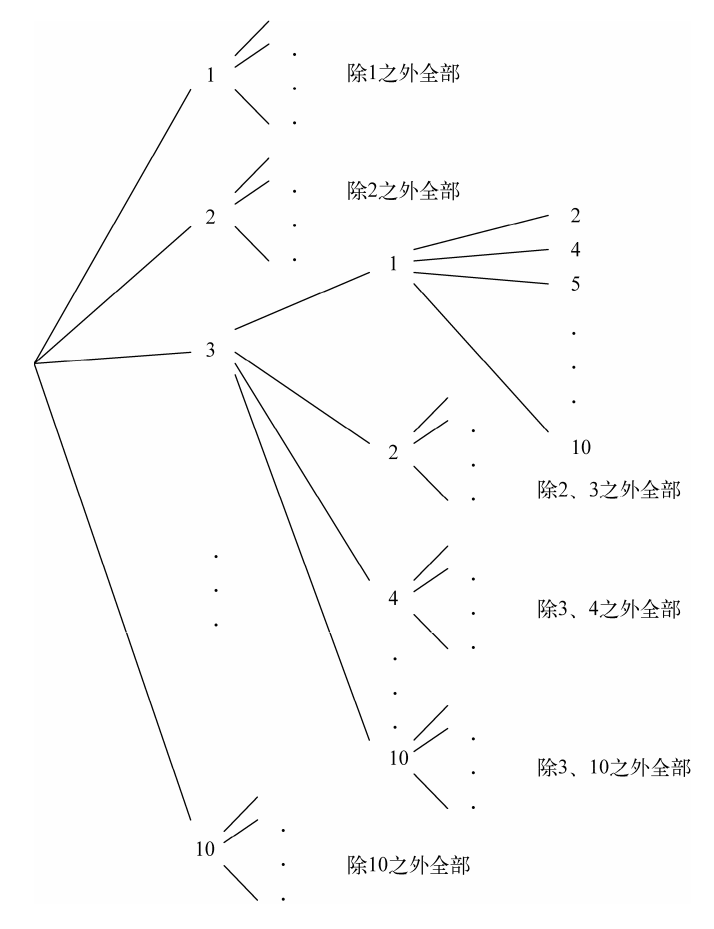


图4-6　从10项有序地选出3项的情况

4.4.1　无放回选择的一般规则

现在来推导一下 $\Pi(n,m)$ 的公式。顺着示例4.5的思路，可知第一次选择时有n种选择。不管第一次作出了怎样的选择，都会剩下 $n-1$ 个元素有待选择。因此，第二次选择有 $n-1$ 种不同的方式。前两次选择总共有 $n(n-1)$ 种方式。类似地，进行第三次选择时还剩 $n-2$ 个未选取的项，所以第三次选择共有 $n-2$ 种不同的方式。因此，前三次选择总共可以有 $n(n-1)(n-2)$ 种方式。

继续用这种方式处理，直到作出m次选择。每次选择都比之前一次的选择少一项。结论就是，从n个项中不放回但有次序地选出m个项，总共有

$$\Pi(n,m)=n(n-1)(n-2)\cdots(n-m+1) \tag{4.3}$$

种不同的方式。也就是说，表达式(4.3)是从n开始依次倒数的m个整数的积。

还可以将(4.3)式写为$n!/(n-m)!$。也就是

$$\frac{n!}{(n-m)!}=\frac{n(n-1)\cdots(n-m+1)(n-m)(n-m-1)\cdots(1)}{(n-m)(n-m-1)\cdots(1)}$$

分母是从1到$n-m$这些整数的积。而分子则是从1到n这些整数的积。因为分子和分母中后$n-m$个因子$(n-m)(n-m-1)\cdots(1)$是相同的，所以将这些项约去，就得到

$$\frac{n!}{(n-m)!}=n(n-1)\cdots(n-m+1)$$

这一公式与(4.3)式是相同的，这样就证实了$\Pi(n,m)=n!/(n-m)!$。

✦ 示例 4.6

考虑一下示例4.5中的情况，其中$n=10$且$m=3$。不难看出，$\Pi(10,3)=10\times9\times8=720$。(4.3)式表示$\Pi(10,3)=10!/7!$，或者说是

$$\frac{10\times9\times8\times7\times6\times5\times4\times3\times2\times1}{7\times6\times5\times4\times3\times2\times1}$$

从1到7这些因数同时出现在分子和分母中，因此要约去这些因数。结果就得到8、9、10这三个数字的积，就是$10\times9\times8$，正如我们在示例4.5中看到的那样。

有放回选择和无放回选择

示例4.5中考虑的问题与4.2节考虑的分配问题只有细微的差别。如果用房屋和颜色来表示，就可以将选出前三名完成比赛的赛马视为将10匹马（颜色）分配给三个完赛排位（房屋）。唯一的区别是，将多所房屋粉刷成相同颜色是可以的，而说一匹赛马同时获得冠军和季军则很荒唐。因此，用10种颜色之一粉刷3所房屋的方法共有10^3或者说是$10\times10\times10$种，而从10匹赛马中选择前三名完成比赛的赛马则有$10\times9\times8$种方法。

有时候我们会将4.2节进行的这种选择称为有放回选择。也就是说，当为一所房屋选择一种颜色（比如说是红色）后，会将红色“放回”可供选择的颜色池中，然后可以继续为其他房屋再次选择红色。

另一方面，我们在示例4.5中讨论的有序选择被称为无放回选择。这种情况下，如果赛马“硬面包”被选作冠军，那么它就不能被放回含有亚军和季军的马匹池了。类似地，如果赛马“秘书处”被选为第二名，那么它也就不可能再成为获得季军的马匹了。

4.4.2 习题

(1) 从26个字母中选出m个字母组成序列，如果不允许同一字母出现一次以上，那么有多少种不同的组合方式？分别计算$m=3$及$m=5$的情况。

(2) 在一个有200名学生的班级中，我们希望选出一位会长、一位副会长、一位秘书和一位财务主管。选择这4位干部的方式共有多少种？

(3) 计算如下阶乘之商：(a)$100!/97!$ (b) $200!/195!$。

(4) “珠玑妙算”（Mastermind）这个游戏要求玩家选择一个由一列4个珠子组成的“密码”，每个珠子都可能是红、绿、蓝、黄、白和黑这6种颜色中的一种。

(a) 总共有多少不同的密码？

(b*) 有两个或多个珠子颜色相同的密码有多少种？提示：这个量是(a)小题的答案与另一个易于计

算的量之间的差。

(c) 不含红色珠子的密码有多少种？

(d*) 不含红色珠子而且至少有两个珠子颜色相同的密码有多少种？

(5) * 通过对n的归纳证明，对1和n之间的任意m，有 $\Pi(n,m) = n!/(n-m)!$ 。

(6) * 通过对 $a-b$ 的归纳证明， $a!/b! = a(a-1)(a-2)\cdots(b+1)$ 。

阶乘之商

请注意，一般而言，只要$b<a$， $a!/b!$就是从$b+1$到a这些整数的积。通过计算

$$a\times(a-1)\times\cdots\times(b+1)$$

来计算阶乘之商，要比分别求出每个阶乘的值然后相除更容易，特别是在b不比a小很多的情况下。

4.5　无序选择

在很多情况下，我们希望计算出从一组项中进行选择到底有多少种方法，而其中所选项的顺序倒是无关紧要。按照4.4节中赛马结果示例的说法，我们可能想知道前三名完成比赛的赛马是哪三匹，但不关心到底哪匹马赢得了哪个名次。换句话说，就是想知道从n匹赛马中选出3匹作为前三名完成比赛的马匹，方法有多少种。

✦　示例 4.7

再次假设 $n=10$ 。我们从示例4.5中得知，选择3匹赛马，假设说是A、B和C，分别作为冠亚季军的方式共有720种。然而，我们现在不关心这3匹马完成比赛的具体次序，只是想知道A、B和C这3匹马以某种次序获得了前三名。因此，我们将通过6种不同的方式得到答案“A、B和C是最好的3匹赛马”，分别对应3匹马在前三名中6种不同的排位。可知刚好存在6种方法，因为给3个项排序的方法为 $\Pi(3)=3!=6$ 种。如果还有疑问的话，可以参考图4-7所示的这6种方法。

冠军	亚军	季军
A	B	C
A	C	B
B	A	C
B	C	A
C	A	B
C	B	A

图4-7　3匹马完成比赛的6种顺序

对A、B和C这3匹马来说成立的情况，对任意一组3匹马来说都成立。在为从10匹马中有序选择出3匹马的情况计数时，每一个3匹马构成的组都会刚好按照它们可能形成的所有次序出现6次。因此，如果只需要计算可能为前三名的3匹马的组合数，就还要在 $\Pi(10,3)$ 的基础上除以6。因此，从10匹马中选出3匹作为前三名的马共有 $720/6=120$ 种不同组合。

✦　示例 4.8

再来考虑一下扑克牌型的数量。在扑克牌游戏中，每名玩家都会分到从52张牌中发出的5张。这里不用考虑分到的5张牌究竟是什么顺序，只关心拿到的这5张牌到底是哪5张。要计算分到的5张牌可能有多少种情况，可以先从计算 $\Pi(52,5)$ 开始，也就是从52个对象中有序选择5个对象的情况总数。这一数字是 $52!/(52-5)!$ ，就是 $52!/47!$ ，或者说是 $50\times49\times48=311\,875\,200$ 。

不过，就像示例4.7中跑得最快的3匹马总共可能以 $3! = 6$ 种次序出现那样，任意一组5张牌都可能以 $\Pi(5) = 5! = 120$ 种不同的次序出现。因此，要在不考虑选择次序的情况下考虑可能构成的牌型，就必须用有序选择的次数除以120。结果是共有 $311\,875\,200/120 = 2\,598\,960$ 种不同的牌型。

4.5.1　为组合计数

现在要将示例4.7和示例4.8中介绍的情况一般化，以得出在不考虑选择顺序的情况下计算从n项中选出m项的方法数的公式。这一函数通常可写为 $\binom{n}{m}$，并说成是“n选m”或是“从n个元素中选取m个元素的组合数”。要计算 $\binom{n}{m}$，首先要计算 $\Pi(n,m) = n!/(n-m)!$，也就是从n个事物中有序选择出m个的方法数。然后要根据选出的这m项来为这些有序选择分组。因为这m项可以有 $\Pi(m) = m!$ 种不同的次序，所以这些分组中各含 $m!$ 个成员。要得到无序选择的数目，就必须要用有序选择的数目除以 $m!$，也就是

$$\binom{n}{m} = \frac{\Pi(n,m)}{\Pi(m)} = \frac{n!}{(n-m)! \times m!} \tag{4.4}$$

✦　示例 4.9

回顾一下示例4.8，它用到了公式(4.4)，其中 $n = 52$ Π，$m = 5$。于是有 $\binom{52}{5} = 52!(47! \times 5!)$。如果将47!与52!中的后47个因数约去，并展开5!，就可以写为

$$\binom{52}{5} = \frac{52 \times 51 \times 50 \times 49 \times 48}{5 \times 4 \times 3 \times 2 \times 1}$$

进行简化，就得到 $\binom{52}{5} = 26 \times 17 \times 10 \times 49 \times 12 = 2\,598\,960$。

4.5.2　*n*选*m*的递归定义

如果递归地考虑从n项中选出m项的方法数，就可以得出计算 $\binom{n}{m}$ 的递归算法。

依据。对任意 $n \geqslant 1$，有 $\binom{n}{0} = 1$。也就是说，从n项中选择0项只有一种方式。此外，$\binom{n}{n} = 1$，也就是说，从n项中选择n项的唯一方法就是将它们都选上。

归纳。如果 $0 < m < n$，那么 $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$。也就是说，如果想从$n$项中选出$m$项，可以用以下两种方法中的任一种。

(1) 不选取第一个元素，接着从剩下的 $n-1$ 个元素中选取m个。$\binom{n-1}{m}$ 这项表示的就是这种情况下可能的选择方法数。

(2) 选取第一个元素，然后从剩下的 $n-1$ 个元素中选取 $m-1$ 个元素。$\binom{n-1}{m-1}$ 这项表示的就是这种情况下可能的选择方法数。

顺便提一句，尽管归纳部分的概念应该很明确（先从全选或全不选的最简单情况开始，进而处理选择某些元素的更复杂的情况），不过还是要谨慎起见，说明是对什么量进行归纳。看待这一归纳的方式之一是，将其视为对m和$n-m$二者中较小的那个与n的积进行完全归纳。那么当该积为0，而且归纳是针对该积的较大值进行时，就会发生依据的情况。我们还必须为归纳过程核实，当$0<m<n$时，$n\times\min(m,n-m)$总大于$(n-1)\times\min(m,n-m-1)$以及$(n-1)\times\min(m-1,n-m)$。这一验证过程将留作本节的习题。

这种递归关系通常是用帕斯卡三角形（Pascal's triangle）[①]表示的，如图4-8所示，其中两条边全部由1构成（表示依据），而三角形中每个内部条目都是它左上角和右上角相邻条目之和。那么$\binom{n}{m}$将作为第$(n+1)$行的第$(m+1)$个条目被读取。

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

图4-8　帕斯卡三角形的前几行

✦ 示例 4.10

考虑一下$n=4$且$m=2$的情况。我们在图4-8第5行的第3个条目处找到了$\binom{4}{2}$的值。该条目为6，而很容易验证$\binom{4}{2}=4!/(2!\times 2!)=24/(2\times 2)=6$。

通过公式(4.4)或是上述递归这两种方法计算$\binom{n}{m}$，计算出的自然是相同的值。可以通过诉诸物理推理（physical reasoning）来证实这一点。两种方法计算的都是从n项中无序选择m项的方法数，所以一定会得出相同的值。不过，还可以通过对n的归纳证明这两种方式的等价性。在这里将该证明过程留作本节的习题。

4.5.3　计算$\binom{n}{m}$的算法的运行时间

正如在示例4.9中所见，当我们使用公式(4.4)计算$\binom{n}{m}$时，可以约去分母中的$(n-m)!$和分子中$n!$的后$n-m$个因数，将$\binom{n}{m}$表示为

$$\binom{n}{m}=\frac{n\times(n-1)\times\cdots\times(n-m+1)}{m\times(m-1)\times\cdots\times 1} \tag{4.5}$$

如果m比n小，那么使用上述公式进行计算要比用公式(4.4)计算更快。大体上讲，图4-9中的C语言代码段就是用来完成这一工作的。

第(1)行将c初始化为1，c就成为了结果——$\binom{n}{m}$。第(2)行和第(3)行会给c乘上从$n-m+1$到

① 又称杨辉三角或贾宪三角。——译者注

n的各个整数。然后，第(4)行和第(5)行会依次从c除去从2到m的各个整数。因此，图4-9就实现了(4.5)式中的公式。

要计算图4-9的运行时间，只要注意到第(2) ~ (3)行及第(4) ~ (5)行这两个循环，每个循环都会迭代m次，而且循环体的运行时间都是$O(1)$。因此，运行时间是$O(m)$。

```
(1)      c = 1;
(2)      for (i = n; i > n-m; i--)
(3)          c *= i;
(4)      for (i = 2; i <= m; i++)
(5)          c /= i;
```

图4-9　计算$\binom{n}{m}$的代码

在m接近n而$n-m$很小的情况下，可以交换m和$n-m$的角色。也就是说，可以约去$n!$和$m!$的因数，得到$n(n-1)\cdots(m+1)$，并将其除以$(n-m)!$。该方法给出了(4.5)所示公式的另一种形式，即

$$\binom{n}{m}=\frac{n\times(n-1)\times\cdots\times(m+1)}{(n-m)\times(n-m-1)\times\cdots\times 1} \tag{4.6}$$

同样，存在与图4-9类似的代码段来实现公式(4.6)，而且所花的时间为$O(n-m)$。因为要定义$\binom{n}{m}$就一定有$n-m$和m不大于n，所以不管是哪种方式，$O(n)$都是运行时间的边界。此外，在m接近0或者接近n时，两种方法中更优方法的运行时间都要大大小于$O(n)$。

不过，图4-9有个重大缺陷。它先要计算若干整数的积，然后再将其除以相同数量的整数。因为普通的计算机运算只能处理有限大小的整数（通常，一个整数最大可以达到约20亿），所以图4-9第(3)行计算中间结果的过程可能有溢出整数大小限制的风险。即使是在$\binom{n}{m}$的值足够小，可以在某计算机中表示出来的情况下，也还是可能出现这种情况。

更好的方式是让乘法和除法交替进行。首先乘上n，然后除以m。乘上$n-1$，再除以$m-1$，以此类推。这种方法的问题在于，我们没理由相信每一阶段的计算结果都是整数。例如，在示例4.9中，首先要乘上52并除以5，这个结果就已然不是整数了。因此，在进行任何计算前都需要转换为浮点数。在这里将这一修改留作本节的习题。

让$\binom{n}{m}$一定得出整数的公式

要看出为什么(4.4)、(4.5)和(4.6)这几个式子中多个因数的商一定是整数可能不容易。唯一的简单论证就是诉诸物理推理。这些公式都是计算从n个事物中选取m个的方法数，而这个数字一定是某个整数。

不借助这些公式的物理意义，而从整数的属性来论证这一事实，要难上很多。其实可以通过仔细分析分子和分母中各质数因子数来证明这一事实。拿示例4.9中的表达式当例子。其中分母中有5这个因数，而分子中有5个因数，由于这些因数是连续的，可知其中必有一个能被5整除，而它正好是中间的那个因数——50。因此，分母中的5肯定会被约去。

现在来考虑计算$\binom{n}{m}$的递归算法。可以通过图4-10所示的简单递归函数来实现这一算法。

图4-10中的函数效率不高，因为它调用choose的次数会呈指数级增长。原因就在于当使用

n作为首个参数调用该函数时，往往会在第(6)行用$n-1$作为首个参数进行两次递归调用。因此，可以预见，当n增加1时，调用的次数就会翻倍。而且递归调用的确切次数是很难计算的。原因在于第(4)行和第(5)行的依据情况不仅适用于$n=1$的情况，而对更大的n，会提供值为0或n的m。

下面要证明一个简单但稍显悲观的上界。设$T(n)$是当首个参数为n时图4-10所示程序段的运行时间。可以直接证明$T(n)$是$O(2^n)$。假设a是第(1)行到第(5)行，加上第(6)行涉及调用与返回的部分（不含递归调用本身所花的时间）的总运行时间。然后就可以通过对n的归纳证明下列命题。

```
        /* 对 0 <= m <= n，计算从 n 个元素中选择 m 个的方法数 */
        int choose(int n, int m)
        {
            int n, m;

(1)         if (m < 0 || m > n) {/* 错误的条件 */
(2)             printf("invalid input\n");
(3)             return 0;
            }
(4)         else if (m == 0 || m == n) /* 依据情况 */
(5)             return 1;
            else /* 归纳 */
(6)             return (choose(n-1, m-1) + choose(n-1, m));
        }
```

图4-10　计算$\binom{n}{m}$的递归函数

命题$S(n)$。如果用第一个参数n以及在0和n之间的第二个参数m调用choose，那么该调用的运行时间$T(n)$至多为$a(2^n-1)$。

依据。$n=1$。那么一定有$m=0$或$m=1=n$。因此，依据情况适用于第(4)行和第(5)行，而且没有进行递归调用。第(1)行到第(5)行的时间都包含在a中，因为$S(1)$是说$T(1)$至多为$a(2^1-1)=a$。

归纳。假设$S(n)$成立，也就是有$T(n)\leqslant a(2^n-1)$。要证明$S(n+1)$成立，假设要以$n+1$为首个参数调用choose。那么图4-10所示程序段花的时间就是a加上第(6)行两次递归调用的时间。根据归纳假设，每次调用花费的时间至多为(2^n-1)。因此，消耗的总时间最多是：

$$a+2a(2^n-1)=a(1+2^{n+1}-2)=a(2^{n+1}-1)$$

这一计算过程就证明了$S(n+1)$成立，并证明了归纳步骤。

因此证明了$T(n)\leqslant a(2^n-1)$。舍去常数因子及低阶项，就可以得出$T(n)$是$O(2^n)$的结论。

奇怪的是，尽管在第3章的分析中，很容易就证明了运行时间的平滑紧上界，但$T(n)$上的边界$O(2^n)$虽平滑却不紧凑。合适的平滑紧上界要稍小一些——$O(2^n\sqrt{n})$。要证明这一事实相当困难，不过在这里要留一个更为简单的事实作为习题来证明，就是图4-10所示程序段的运行时间与它返回的值$\binom{n}{m}$成比例。要看到图4-10中的递归算法，效率要比图4-9中的算法低得多。这是一个递归严重不靠谱的例子。

4.5.4　$\binom{n}{m}$函数的图像

对某个固定的值n而言，m的函数$\binom{n}{m}$有着不少有意思的属性。对于值比较大的n来说，如

图4-11所示，其图像为一条钟形的曲线。我们很容易看出函数图像是关于中点 $n/2$ 所在轴线对称的，运用声明 $\binom{n}{m}=\binom{n}{n-m}$ 的公式(4.4)很容易证实这一点。

最大高度处于中心位置，也就是 $\binom{n}{n/2}$ ，大约是 $2^n/\sqrt{\pi n/2}$ 。例如，若 $n=10$ ，这一公式可以得出258.37，而 $\binom{10}{5}=252$ 。

该曲线的"厚部"是中点两边各约 $\sqrt{n}$ 的范围。例如，如果 $n=10\ 000$ ，那么对处在4900和5100之间的m，$\binom{10000}{m}$ 就接近最大值。而对这个范围之外的m来说，$\binom{10000}{m}$ 的值会下降得特别迅速。

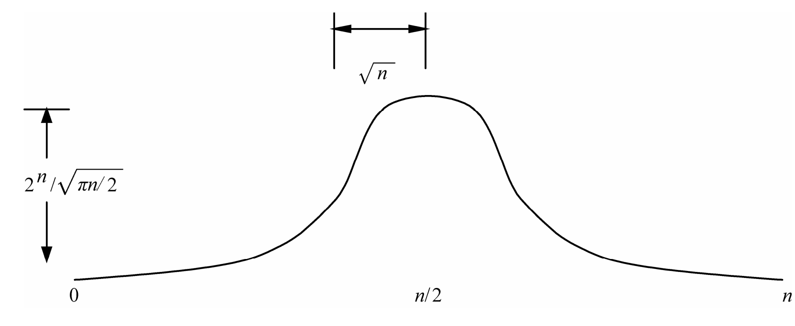


图4-11 n为固定值的 $\binom{n}{m}$ 函数

4.5.5 二项式系数

函数 $\binom{n}{m}$ 除了可以用来计数外，还能提供*二项式系数*。在展开二项式的乘方（比如 $(x+y)^n$ ）时，就会看到这些数字。

在展开 $(x+y)^n$ 时，会得到2^n个项，其中每一项都是 $x^m y^{n-m}$ 这样的形式（m是0到n之间的某个整数）。也就是说，对每个因式 $x+y$ ，都可能从x和y中任选其一作为某个特定项的因子。展开式中 $x^m y^{n-m}$ 的系数是由m个x和其余 $n+m$ 个y组成的项的数量。

✦ 示例 4.11

考虑一下 $n=4$ 的情况，也就是看看 $(x+y)(x+y)(x+y)(x+y)$ 的积。

总共有16项，其中只有1项是 x^4y^0 （也就是 x^4 ）。如果从4个因式中都选出x，就能得到这一项。另一方面，有4项是 x^3y ，对应的情况是从4个因式的任意一个中选出y，再从其余3个因式中选出x。对称地，有1项是 y^4 ，有4项是 xy^3 。

那么有多少项是 x^2y^2 呢？如果从两个因式中选取x并从其余两个中选取y，就能得到这样一项。因此，必须要计算从4个因式中选择两个因子的方法数。因为选择两个因子的顺序是不产生影响的，所以这个数字就是 $\binom{4}{2}=4!/(2!\times 2!)=24/4=6$ 。因此，有6项是 x^2y^2 。完整的展开式就是

$$(x+y)^4=x^4+4x^3y+6x^2y^2+4xy^3+y^4$$

请注意等式右侧各项的系数，(1, 4, 6, 4, 1)，正好就是图4-8中帕斯卡三角形的一行。我们会看

到，这并非巧合。

示例4.11中用于计算 x^2y^2 系数的概念可以推广开来。$(x+y)^n$ 展开式中的项 $x^m y^{n-m}$ 的系数为 $\binom{n}{m}$。原因在于，只要从n个因式中选出m个x并选出 $n-m$ 个y，就可以得到 $x^m y^{n-m}$ 这一项。从n个因式中选出m个因子的方式有$\binom{n}{m}$种。

在二项式系数和$\binom{n}{m}$函数之间还有一种有趣的关系。我们已经看出

$$(x+y)^n = \sum_{m=0}^{n} \binom{n}{m} x^m y^{n-m}$$

令 $x=y=1$，那么有 $(x+y)^n = 2^n$。x和y的所有乘方都是1，所以上述等式就成了

$$2^n = \sum_{m=0}^{n} \binom{n}{m}$$

换句话说，对某个特定的n而言，所有二项式系数的和就是2^n。特别要说的是，每个系数$\binom{n}{m}$都小于 2^n。图4-11就暗示了，对接近 $n/2$ 的m来说，$\binom{n}{m}$和 2^n 特别接近。由于在图4-11中曲线下方的区域表示 2^n，因此能看出为什么只有接近中点的一些值会比较大。

4.5.6　习题

(1) 计算以下各值：(a) $\binom{7}{3}$；(b) $\binom{8}{3}$；(c) $\binom{10}{7}$；(d) $\binom{12}{11}$。

(2) 从26个小写字母中选出5个不同字母的方法共有多少种？

(3) 如下系数各为多少？

　(a) $(x+y)^7$ 的展开式中 x^3y^4 的系数；

　(b) $(x+y)^8$ 的展开式中 x^5y^3 的系数。

(4) * 在Real Security公司，计算机密码必须由4位数字（10选4）和6个字母（52选6）组成，字母和数字都可以重复。总共可能有多少种不同的密码组合？提示：首先考虑选择4个存放数字的位置共有多少种方法。

(5) * 5个字母组成的双元音序列有多少种？

(6) 重新编写图4-9所示的程序片段，从而利用 $n-m$ 小于n的情况。

(7) 重新编写图4-9所示的程序片段，并将其转换成浮点数乘除交替的算法。

(8) 证明：如果 $0 \leqslant m \leqslant n$，那么 $\binom{n}{m} = \binom{n}{n-m}$。

　(a) 借助$\binom{n}{m}$的含义。

　(b) 利用(4.4)式。

(9) * 通过对n的归纳，证明$\binom{n}{m}$的递归定义正确地定义了$\binom{n}{m}$等于 $n!/\big((n-m)!\times m!\big)$。

(10) ** 通过对n的归纳，证明图4-10中递归函数`choose(n,m)`的运行时间最多是$c\binom{n}{m}$，其中c为某个常数。

(11) * 证明：当 $0<m<n$ 时，$n\times\min(m,n-m)$ 总是大于 $(n-1)\times\min(m,n-m-1)$ 和 $(n-1)\times\min(m-1,n-m)$。

4.6　相同项的次序

在本节中，要处理的是这样一些选择问题，其中含有一些相同项，但不同项出现的次序很重要。而在接下来的4.7节中，则要解决一类类似的选择问题，即有一些相同项，而且项的次序无关紧要。

✦ 示例 4.12

构词（anagram）猜谜游戏会给出一列字母，让玩家重新排列字母以构成单词。如果拥有含规范单词的字典，并能生成所有可能的字母序列，就可以通过计算机解决该问题。第10章会介绍判定给定字母序列是否处于字典中的有效方法。不过现在要考虑的是组合问题，可能首先要确定有多少单词需要用字典验证其确实存在。

对有些构词来说，计数很简单。假设有`abenst` 6个字母，可能会有 $\Pi(6)=6!=720$ 种不同的次序，其中之一便是`absent`，也就是该谜题的"解答"。

不过，构词游戏通常会含有重复的字母。考虑一下谜题`eilltt`。这些字母就不能构成720种不同的序列。例如，交换两个字母`t`的位置似乎并不能让单词发生变化。

假设对两个`t`和两个`l`加以标记以区分这些字母，分别将其记为t_1、t_2、l_1和l_2。被标记的字母可能有720种次序。然而，这些标记过的l仅在位置上有区别，诸如$l_2it_2t_1l_1e$和$l_1it_2t_1l_2e$就并不是真的有区别。因为所有720种次序可以平分为两组，这两组的区别只在于`l`的下标，所以可以证明：如果将字母串的数量除以2，这些`l`其实都是相同的。

类似地，在字符串中只有字母`t`带标记时，可以将只有`t`的下标不同的字符串配对。例如，lit_1t_2le和lit_2t_1le就是一对。因此，如果再将数目除以2，就可以得到将`t`和`l`的标记删除后不同构词串的数量。该数字为360/2=180。即使用`eilltt`共有180种不同的构词方法。

我们可以将图4-12中的概念一般化为有n个项而且这些项被分为k组的情形。各组中的成员都是相同的，而不同组的成员则是不同的。在这里假设m_i是第i组中的成员项，其中 $i=1,2,\cdots,k$。

✦ 示例 4.13

重新考虑示例4.12中用`eilltt`构词的问题。其中共有6项，也就是说 $n=6$。而分组的数量k为4，因为有4个不同的字母。这4组中有两组含有一个成员（e和i），而另两组则含两个成员。因此可以取 $i_1=i_2=1$，$i_3=i_4=2$。

如果为这些项加上标记，以使同一组中的成员有所不同，那么会有$n!$种不同的次序。不过，若第一组中有i_1个成员，那么这些标记过的项可能会以$i_1!$ 种不同的次序出现。因此，在从第一组的项上移除标记时，我们要将这些次序分成大小同为$i_1!$的集合。因此必须将次序数除以$i_1!$，从而得到从第1组删除标记后的次序数。

类似地，依次从各组中删除标记需要将不同次序的数量除以$i_2!$、除以$i_3!$，等等。对那些值为1的$i_j!$ 来说，就是除以$1!=1$，因此没有任何影响。不过，对那些所含项数大于1的分组来说，我们必须除以分组大小的阶乘，这就是示例4.12中的情况。有两组中包含1个以上的元素，而每组的大小都是2，所以就要除以2!两次。可以通过对k的归纳证明该一般规则。

命题 $S(k)$。如果有n个项，并且分别被分为大小为i_1、i_2、$\cdots$、i_k的k个组，同一组中的项是相同的，而不同组中的项是不同的，那么这n个项能形成的不同次序的数目为

$$\frac{n!}{\prod_{j=1}^{k} i_j!} \tag{4.7}$$

依据。如果 $k=1$，那么只有一组无区别的项，不管n有多大都只有一种次序。如果 $k=1$，那么 i_1 一定为n，而(4.7)式就变成了 $n!/n!$，也就是1。因此，$S(1)$ 成立。

归纳。假设 $S(k)$ 为真，并考虑有 $k+1$ 个分组时的情况。设最后一组中有 $m=i_{k+1}$ 个成员。这些项将会出现在m个位置，而且可以有 $\binom{n}{m}$ 种不同的方式来选择这些位置。一旦选定了m个位置，把最后一组中的哪一项放在这些位置都没关系了，因为这些项都是没有区别的。

在为最后一组选择了位置后，还剩 $n-m$ 个位置来容纳其余k个组。归纳假设适用，并且表明最后一组的每种位置选择都对应着其余位置中其余元素的 $(n-m)!/\prod_{j=1}^{k} i_j!$ 种不同次序。该式与(4.7)式相比，只是将(4.7)式中n的位置替换成了 $n-m$，因为只剩 $n-m$ 项有待放置了。因此 $k+1$ 组项的次序总数为

$$\frac{\binom{n}{m}(n-m)!}{\prod_{j=1}^{k} i_j!} \tag{4.8}$$

如果将(4.8)中的 $\binom{n}{m}$ 替换成等价的 $n!\big((n-m)!m!\big)$，就得到

$$\frac{n!}{(n-m)!m!}\frac{(n-m)!}{\prod_{j=1}^{k} i_j!} \tag{4.9}$$

可以从(4.8)式的分子和分母中约去 $(n-m)!$。此外，请记住m是 i_{k+1}，是第 $k+1$ 组中的成员的数目。因此可得到次序数为

$$\frac{n!}{\prod_{j=1}^{k+1} i_j!}$$

这正是 $S(k+1)$ 所给出的式子。

✦ 示例 4.14

一位探险家带了两个星期的口粮，其中包括4罐金枪鱼、7罐午餐肉以及3罐黄豆罐头。如果他每天打开一罐罐头，那么他消耗这些口粮的次序共有多少种？这里的14项分成了分别具有4、7和3个相同项的3组。按照(4.7)式，其中 $n=14$，$k=3$，$i_1=4$，$i_2=7$，$i_3=3$。因此他消耗口粮的次序数为

$$\frac{14!}{4!\times 7!\times 3!}$$

先从分母中的7!开始，可以约去分子14!中最后7个因数。因此就得到

$$\frac{14\times 13\times 12\times 11\times 10\times 9\times 8}{4\times 3\times 2\times 1\times 3\times 2\times 1}$$

继续约去分子和分母中的因数，就可以得到结果为120 120。也就是说，消耗这些口粮的方法有逾10万种。可惜每一种听起来都让人没什么胃口。

习题

(1) 计算以下单词的字母构词的数量：(a) `error`；(b) `street`；(c) `allele`；(d) `Mississippi`。

(2) 将下列水果排成一线共有多少种方法？
　(a) 3个苹果、4个梨和5根香蕉；
　(b) 2个苹果、6个梨、3根香蕉和2颗李子。
(3) * 将白王、黑王、2个白骑士和1个黑车摆在棋盘上，共有多少种摆法？
(4) * 100个人参与到一场彩票游戏中。其中一人可赢得千元大奖，还有5人可以得到50美元储蓄基金的安慰奖。那么总共可能有多少种不同的获奖结果？
(5) 写一个简单的公式，用来计算放置n对两两相等的$2n$个对象的次序数。

4.7　将对象分装入箱

我们要介绍的下一类计数问题涉及对盛装若干对象之容器的选择。这些对象可能相同，也可能不同，不过容器是有区别的。我们必须计算这些装满容器的方法数。

✦　示例 4.15

有凯西、彼得和苏姗3个孩子，我们要将4个苹果分给他们，而不把苹果切开。那么共有多少种分配苹果的方式？

这里的方法数比较少，因此可以直接将其枚举出来。凯西可能得到从0至4个不等的苹果，而不管余下几个苹果，分给彼得和苏姗的方式都只有少数几种。如果设(i, j, k)表示凯西得到i个苹果、彼得得到j个苹果而苏姗得到k个苹果的情况，那么图4-12就展示了全部15种可能的分配方式。每一行对应着分给凯西的苹果数。

(0，0，4)	(0，1，3)	(0，2，2)	(0，3，1)	(0，4，0)
(1，0，3)	(1，1，2)	(1，2，1)	(1，3，0)	
(2，0，2)	(2，1，1)	(2，2，0)		
(3，0，1)	(3，1，0)			
(4，0，0)				

图4-12　把4个苹果分给3个孩子共有15种方式

为将相同对象分装入箱计数的方法有个诀窍。假设用4个字母A来表示4个苹果，并用两个*来分隔属于不同孩子的苹果。两个*之间的A的数量就表示彼得分到的苹果数，而第二个*之后的A的数量则表示属于苏姗的苹果数。例如，AA*A*A表示（2,1,1）的分配方式，其中凯西分到2个苹果，其余两个孩子各分到1个。而序列AAA*A*则表示（3,1,0）的分配方式，其中凯西得到3个，彼得得到1个，苏姗一个都没有。

因此，每种分发苹果的方式都与由4个A和2个*组成的唯一字符串相关。那么有多少这样的字符串呢？考虑一下组成这种字符串的6个位置。其中任选4个位置用来存放A，另外两个位置用来存放*。正如我们在4.5节中了解到的，从6项中选择4项共有$\binom{6}{4}$种方法。因为$\binom{6}{4}=15$，所以又一次得出了将4个苹果分给3个孩子的方法共有15种的结论。

4.7.1　装箱问题的一般规则

我们可以按照下列方式将示例4.15介绍的问题一般化。假设给定n个容器，它们对应示例中的3个孩子。同时假设要将m个相同的对象随意地放进这些容器中。那么有多少种分装入箱的方式呢？

这里可以再次考虑A和*组成的字符串。A表示对象，而*表示容器间的边界。如果有n个对象，就有n个A，而如果有m个容器，那么就需要$m-1$个*来表示分隔不同容器的边界。因此，字符串的长度为$n+m-1$。

我们可以从这些位置中任选n个存放A，剩下的就是存放*的。因此共有$\binom{n+m-1}{n}$种由A和*组成的字符串，那么将对象分装入箱的方式也有这么多种。在示例4.15中，有$n=4$且$m=3$，所以就可以得到共有$\binom{n+m-1}{n}=\binom{6}{4}$种分配方式的结论。

✦　示例 4.16

在掷骰子游戏中，要掷出3个骰子，其中每个骰子的6个面上都标记了从1到6这6个数字。玩家可以为某个数字赌上1美元。如果这个数字不出现，钱就输掉了。如果该数字出现一次或多次，那么该玩家就可以得到与该数字出现次数等额的美元。

我们可能想要为“结果”计数，不过一开始在“结果”是什么的问题上可能有些疑问。如果将骰子不同的面涂上不同颜色以方便区别，就可将其视为4.2节中那样的计数问题，其中3颗骰子中的每一颗都能分配6个数字中的一个。我们知道，进行这样的分配共有$6^3=216$种方式。

不过，骰子通常是没有区别的，这些数字出现的顺序也是无关紧要的，只有每个数字出现的次数决定了哪个玩家会赢钱，会赢多少钱。例如，掷骰子的结果可能是有两颗是1，而第三颗是6。而6可能出现在第1颗、第2颗或第3颗骰子上，不过出现在哪颗骰子上都是没关系的。

因此，可以把这一问题视为将相同对象分装入箱的问题。“容器”就是1到6这几个数字，而“对象”就是3个骰子。一颗骰子会被“分装”到对应该骰子掷出数字的那个容器。因此，掷骰子游戏总共有$\binom{6+3-1}{3}=\binom{8}{3}=56$种不同的结果。

4.7.2　分装有区别的对象

我们可以将之前的公式扩展一下，以便处理将可分为k类的n个对象装入m个容器的问题。同一类中的对象是没有区别的，但不同类的对象是不同的。这里用符号a_i表示第i类中的成员。因此可以构成由下列对象组成的字符串。

(1) 对每个类i，与类中所含成员数量等量的a_i；

(2) 用来表示m个容器间的边界的$m-1$个*。

因此这些字符串的长度是$n+m-1$，请注意，这些*构成了第k+1个类，而该类包含了m个成员。

我们在4.6节中已经了解过如何为这样的字符串计数。字符串的个数为

$$\frac{(n+m-1)!}{(m-1)!\prod_{j=1}^{k} i_j!}$$

其中i_j表示的是第j类中的成员数。

✦　示例 4.17

假设有三个苹果、两个梨和一根香蕉要分给凯西、彼得和苏姗。那么“容器”的数量，也就是孩子的数量$m=3$。共有$k=3$组，分别有$i_1=3$、$i_2=2$和$i_3=1$个成员。因为总共有6个对象，所以n=6，因此该问题中的字符串的长度为$n+m-1=8$。这些字符串由3个表示苹果的`A`、两个表示梨的`P`、一个表示香蕉的`B`，以及两个表示边界的`*`组成。因此，由分发方法数的计算公式可得到共有

$$\frac{(n+m-1)!}{(m-1)!i_1!i_2!i_3!}=\frac{8!}{2!3!2!1!}=1680$$

种将这些水果分发给凯西、彼得和苏姗的方式。

计数问题的对比

在本节及之前的4.1到4.5节中，我们已经考虑了6种不同的计数问题。每种问题都可视作为特定的位置分配对象。例如，4.2节介绍的分配问题可以视为给定了n个位置（对应房屋），以及不限量的具有k个不同类型（对应颜色）之一的对象。我们可以顺着3个方向为这些问题分类。

(1) 它们是否会放置所有给定的对象？

(2) 分配对象的次序是否重要？

(3) 所有对象都是不同的，还是说某些对象没有区别？

下表表示了之前各节提到的问题之间的区别。

节	典型问题	是否必须使用所有对象	次序是否重要	是否有相同的对象
4.2	粉刷房屋	否	是	否
4.3	排序	是	是	否
4.4	赛马比赛	否	是	否
4.5	扑克牌型	否	否	是
4.6	构词	是	是	是
4.7	给孩子分苹果	是	否	是

4.2节和4.4节中的问题在上表中体现不出什么区别。它们的区别在于是否放回，正如之前4.4.1节附注栏“有放回选择和无放回选择”中讨论的那样。也就是说，在4.2节中，每种“颜色”都是不限量供应的，可以多次选择同一颜色。而在4.4节中，一匹被选定的“赛马”不能在同一系列的选择中再被选中了。

4.7.3 习题

(1) 进行下列分配任务分别有多少种方法？

(a) 6个苹果分给4个孩子；

(b) 4个苹果分给6个孩子；

(c) 6个苹果和3个梨分给5个孩子；

(d) 2个苹果、5个梨和6根香蕉分给3个孩子。

(2) 下列情况分别有多少种结果？

(a) 掷4颗无区别的骰子；

(b) 掷5颗无区别的骰子。

(3) * 将7个苹果分给3个孩子，并保证每个孩子至少得到一个苹果，共有多少种分发方法？

(4) * 假设从国际象棋棋盘的左下角开始向右上角移动，每次向上或向右移动一格，完成这一移动的方式共有多少种？

(5) * 将习题(4)一般化。如果有一个由n个方格乘上m个方格组成的矩形，并可以从一个方格向上或向右移动到另一个方格，那么从左下角移动到右上角总共有多少种方法？

4.8 计数规则的组合

组合这一主题能带来无数的挑战，而且很少像本章之前所讨论的那样简单。不过，我们目前所了解到的规则都是最基础，它们都很有价值，能以各种方式结合起来为更加复杂的结构计

数。在本节中，我们将了解到3种实用的计数“诀窍”。

(1) 将计数表示为一系列选择；

(2) 将计数表示为计数的差；

(3) 将计数表示为子情况的计数之和。

4.8.1　将计数分解为一系列选择

在为某类分配计数时，有一种实用的方法可以采用，就是将这些待计数的事物描述为一系列的选择，其中每次选择都会细化该类中某个特定成员的描述。在本节中，我们会给出一系列表示某些可能性的示例。

✦ 示例 4.18

考虑一下扑克牌型中“对子”（one-pair）的数量。该牌型由一对具有某个秩[①]的牌，加上3张具有不同秩（而且与之前一对的秩不同）的牌组成。我们可以通过如下步骤描述所有的“对子”牌型。

(1) 为成对的牌选择秩；

(2) 从其余12种秩中为其余3张牌选择3个不同的秩；

(3) 为成对的牌选择花色；

(4) 为其他3张牌选择花色。

如果将这些数字相乘，将得到“对子”牌型的数量。请注意，牌型中各张牌出现的顺序是无关紧要的，正如之前在示例4.8中讨论过的，而且我们从未尝试过指定次序。

现在，要依次接受这些因素。为成对的那两张牌选择秩的方式有13种。不管选择了哪个秩，都会余下12种。接下来必须从这些秩中选择3个组成剩下的牌型。就像4.5节中讨论过的，这也是一种次序不重要的选择，执行这种选择的方式共有$\binom{12}{3}=220$种。

现在必须为这对牌选择花色。共有4种花色，而且我们必须从中选择两种。这次又是无序的选择，可以有$\binom{4}{2}=6$种方式。最后，为剩下的3张牌选择花色。每张牌都有4种花色可选，所以又是4.2节中那样的分配问题，进行分配的方式共有$4^3=64$种。

因此，“对子”牌型的总数量为$13\times220\times6\times64=1\,098\,240$种，这一数字在2 598 960种扑克牌型中占了40%以上。

4.8.2　用计数的差来计算计数

另一种实用技巧是，将要计数的内容表示为某个更具一般性的排列类C与C中不满足计数条件的那些事物之间的差。

✦ 示例 4.19

还有很多种扑克牌型（两对、三条、铁支和葫芦）可以按照类似示例4.18的方法计数。不过，还有一些其他牌型需要不同的方法来计数。

先来考虑一下同花顺的情况，也就是5张花色相同（同花）而且秩连续（顺子）的牌型。首先，每个顺子都是从A到10这10个秩之一开始的。也就是说，顺子可能是A-2-3-4-5，2-3-4-5-6，3-4-5-6-7，等等，最大的可能是10-J-Q-K-A。一旦秩确定，就只需要指定一种花色来指定该同

---

① 13个秩分别为A、K、Q、J，以及2到10。

花顺了。因此，为同花顺计数包含以下两步：

(1) 选择顺子的最低秩（10种选择）；

(2) 选择花色（4种选择）。

因此，总共有$10 \times 4 = 40$种同花顺牌型。

现在来为顺子牌型计数，也就是那些秩连续但又不是同花顺的牌型。先要计算所有具有连续秩的牌型的数量，不考虑它们的花色是否相同，然后再减去40种同花顺牌型。要为秩连续的牌型计数，可以按以下两步进行：

(1) 选择最低的秩（10种选择）；

(2) 为每个秩指定一种花色（如4.2节介绍的，有$4^5 = 1024$种选择）。

因此，顺子和同花顺牌型的总数是$10 \times 1\,024 = 10\,240$种。减去40种同花顺牌型后，就得出顺子牌型共有$10\,240 - 40 = 10\,200$种。

接下来，考虑一下同花牌型的数目。这里还是要先将同花顺牌型考虑在内，然后再减去那40种同花顺牌型。可以通过如下方式定义同花牌型。

(1) 选择花色（4种选择）；

(2) 从13个秩中任选5个，如4.5节介绍的，共有$\binom{13}{5} = 1\,287$种方式。

于是可以得出同花牌型共有$4 \times 1\,287 - 40 = 5\,108$种。·

4.8.3 将计数表示为子情况的和

在面对一些很难直接解决的问题时，就要用到第三种“诀窍”了。可以把为某个类C计数的问题分解成两个或多个单独的问题，而类C中的各个成员刚好都能被子问题之一涵盖。

✦ 示例 4.20

假设要抛10次硬币，那么8个或8个以上硬币人头面朝上的序列有多少种？如果想知道有多少序列刚好有8个硬币人头面朝上，可以用4.5节介绍的方法来解决。共有$\binom{10}{8} = 45$种这样的序列。

要解决为8个或8个以上硬币人头面朝上的序列计数的问题，可以将其分解为3个子问题，即分别为刚好有8个硬币人头面朝上、刚好有9个硬币人头面朝上以及10个硬币全部人头面朝上的情况计数。我们已经解决了第一个问题。而9个硬币人头面朝上的序列共有$\binom{10}{9} = 10$种，10个硬币全是人头面朝上的序列共有$\binom{10}{10} = 1$种。因此，有8个或8个以上硬币人头面朝上的序列共有$45 + 10 + 1 = 56$种。

✦ 示例 4.21

再来考虑一下示例4.16中解决的为掷骰子游戏的结果计数的问题。另一种方法就是根据所出现的不同数字是3个、2个或1个，将该问题分成3个子问题。

(a) 可以用4.5节介绍的技巧计算3个数字皆不同的结果的数量。也就是从一颗骰子6个可能的数字中选出3个，总共有$\binom{6}{3} = 20$种不同的方法。

(b) 接着，要计算两颗骰子是一个数，而另一颗是另一个数的情况有多少种。出现两次的数字有6种选择，每种情况下对应的出现一次的数字有5种选择。所以两颗骰子是一个数而另一颗

是另一个数的结果共有 $6\times5=30$ 种。

(c) 3颗骰子数字全相同的情况共有6种。

因此，可能的结果共有 $20+30+6=56$ 种，这与示例4.16中得到的结论是一样的。

4.8.4 习题

(1) * 为以下扑克牌型计数：

(a) 两对；

(b) 三条；

(c) 葫芦（三条加一对）；

(d) 铁支（四条）。

请注意，在为某种牌型计数时不要将那些更佳的牌型计算在内了。例如，在情况(a)中，要确定两对是不同的，不然就是拿到了铁支牌型，而且要保证第5张牌和这两对的秩不一样，否则就是拿到了葫芦牌型。

(2) * *黑杰克*由两张牌组成，其中一张是A，而另一张是10分牌，就是10、J、Q或K中的一种。

(a) 在一摞52张扑克牌中，有多少种不同的黑杰克？

(b) 在黑杰克游戏中，有一张牌是暗牌，而另一张是明牌。因此，两张牌的次序是有影响的。在这种情况下，有多少种不同的黑杰克？

(c) 在皮诺奇勒牌游戏中，只使用秩为9、10、J、Q、K和A的牌各8张（每种花色各有两张同秩的牌），而不使用其他扑克牌。假设次序无关紧要，共有多少种黑杰克？

(3) “什么都不是”（即不是对子或更好）的牌型共有多少种？大家可能要利用示例4.18和示例4.19的结果以及习题(1)的解答。

(4) 如果依次抛12枚硬币，那么下列情况各有多少种？

(a) 至少有9枚硬币人头面朝上；

(b) 至多有4枚硬币人头面朝上；

(c) 有5到7枚硬币人头面朝上；

(d) 不到2枚或多于10枚硬币人头面朝上。

(5) * 至少有一个数字为1的掷骰子结果共有多少种？

(6) * 使用单词`little`的字母构词，其中两个`t`不相邻的情况共有多少种？

(7) ** 桥牌牌型由52张扑克牌中的13张构成，我们通常会通过“分布”为牌型分类，也就是说，会按照花色为手牌分组。例如，牌型4-3-3-3的分布表示有4张牌是某种花色，而另外3种花色的牌各有3张。牌型5-4-3-1的分布表示各花色的牌分别有5张、4张、3张和1张。为具有如下分布的牌型计数：(a)4-3-3-3；(b)5-4-3-1；(c)4-4-3-2；(d)9-2-2-0。

4.9 概率论简介

概率论在计算机科学领域具有很多用途，其中一种重要应用就是估算平均输入或典型输入情况下的程序运行时间。这种估算对那些最坏情况运行时间远大于平均运行时间的算法来说非常重要。我们很快就会看到这种估算的示例。

概率的另一种用途是在具有不确定性的情况下设计制定决策的算法。例如，可以使用概率论，设计根据可用信息制定最佳医疗诊断的算法，或设计在未来预期需求的基础上分配资源的算法。

4.9.1 概率空间

概率空间是指点的有限集，这些点分别表示某一实验的某种可能结果。每个点x都与某个称

作x的概率的正实数相关联，而所有点对应概率的和为1。还有无限多个点的概率空间这样的说法，不过这种概念在计算机科学领域鲜有应用，所以这里不需要考虑这些。

通常，概率空间中的点都是等可能的。除非特别声明，否则可以假设概率空间中若干点表示的概率都是相等的。因此，如果概率空间中有n个点，则每个点的概率都是$1/n$。

✦ 示例 4.22

图4-13所示为具有6个点的概率空间。这些点分别被标记为从1到6这6个数字中的一个，而且可将该空间视为表示掷一次骰子这项“实验”的结果。也就是说，骰子6个面中某个面上的数字会出现在最上方，而每个数字出现的概率都是均等的，即都是$1/6$。

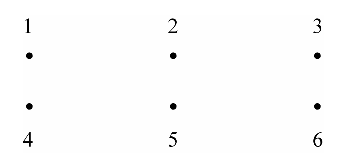


图4-13　具有6个点的概率空间

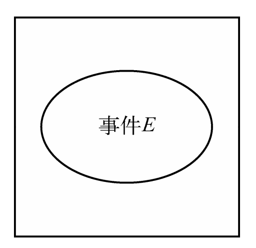
概率空间中这些点的任意子集都可称为事件。某个事件E的概率——记为PROB(E)——就是E中各点概率之和。如果这些点都是等可能的，就可以用E中点的数目除以整个概率空间中点的数目来计算E的概率。

4.9.2　概率的计算

通常情况下，计算某个事件的概率会涉及组合。我们必须计算事件中点的数量，以及整个概率空间中点的数量。当点是等可能时，这两个计数的比就是该事件的概率。接下来要介绍一系列示例，展示按这种方式计算概率的过程。

无限的概率空间

在某些情况下，可以想象一个具有无数个点的概率空间，其中任何给定的点的概率都可能是无穷的，从而只能将有限的概率与某些点的集合关联起来。举个简单的例子，下图中的正方形表示某个概率空间，而该概率空间中的点是该正方形所在平面上的所有点。



可以假设正方形中任何点被选中的可能性都是相等的，并将这种“实验”视作往该正方形中投掷飞镖，飞镖飞到正方形上任何位置的可能性都是相同的，但肯定不会飞到正方形之外。虽然任何一点被击中的概率都是无穷的，但是该正方形中某个区域的概率等于该区域的面积与整个正方形的面积之比。因此，我们可以计算某些事件的概率。

例如，上图的概率空间中含有一个椭圆形组成的事件E。假设该椭圆区域的面积是整个正方形面积的29%，那么PROB(E)就是0.29。也就是说，如果随机地向该正方形投出飞镖，那么29%的情况下飞镖会落在这个椭圆中。

✦ 示例 4.23

图4-14展示了表示掷两颗骰子的概率空间。也就是说，进行的实验是按顺序抛出两颗骰子，并观察它们朝上那面的数字。假设掷骰子的过程是公平的，就有36个等可能的点，或者说是实验结果，所以每个点的概率都是1/36。每个点对应着每颗骰子从6个值中任选其一的分配。例如，(2,3)就表示第一颗骰子为2点，而第二颗骰子为3点的情况。而(3,2)则表示第一颗骰子是3，第二颗是2的情况。

被圈出来的区域表示"吃憋"的事件，也就是两颗骰子的总点数为7或11的情况。这个事件共含有8个点，其中6个是总点数为7的情况，而有两个是总点数为11的情况。投出"吃憋"情况的概率就是8/36，约为22%。

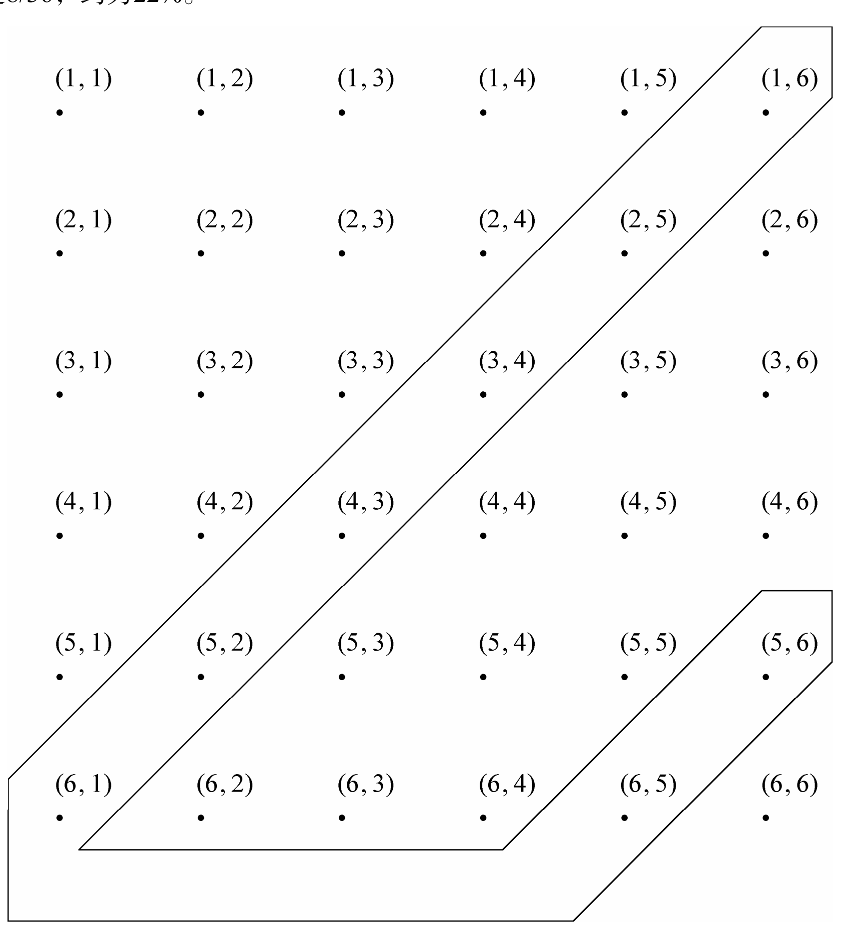


图4-14　掷两颗骰子的概率空间中的"吃憋"事件

✦ 示例 4.24

再来计算一下扑克牌出现对子牌型的概率。我们在示例4.8中已了解到总共有2 598 960种不同的扑克牌型。考虑该公平处理扑克牌型的实验，也就是说，所有牌型出现的可能性都相等。因此，该实验的概率空间总共有2 598 960个点。我们还从示例4.18中得知，这些表示牌型的点中有1 098 240个被分类为对子牌型。假设所有牌型被处理的可能都是均等的，那么"对子"事件的概率就是1 098 240/2 598 960，大约是42%。

✦ 示例 4.25

在基诺游戏中，会随机从1到80这些数字中选出20个。而且选取这些数字之前，玩家可以猜测一些数字。这里要专门讲讲这个玩家要猜测5个数字的“5点游戏”。所猜数字与选中的20个数字中有3个、4个或5个吻合的话，玩家就中奖了，而且猜中的数字越多，得到的奖金越丰厚。我们要计算的是玩家在该游戏中正好猜中3个数字的概率。正好猜中4个或5个数字的概率的计算将留作本节的习题。

首先，合适的概率空间包含了表示从1到80中任选20个数字所有可能情况的点。这样的选择总共有

$$\binom{80}{20}=\frac{80!}{20!60!}$$

种，这个数字奇大无比，好在不需要把它写出来。

结果何时是随机的？

在示例中已经假设过某些实验具有“随机的”结果，也就是说，所有可能出现的结果的可能性都是相等的。在一些情况下，这种假设的合理性源自物理学。例如，在投掷公平（未加重）的骰子时，我们假设在物理上不可能控制骰子的某个面比其他面更可能朝上。这在实践中是种有效的假设。同样，我们可以假设公平洗牌的牌堆不会影响结果，而且任何一张牌出现在牌堆中任意位置的可能性都是相同的。

在其他情况下，我们发现一些貌似随机的事物实际上根本不随机，只不过是某个过程原则上可预知但在实践中不可预知的结果。例如，基诺游戏中选出的数字可能是由计算机执行某个特殊算法生成的，而如果没法接触到计算机使用的这些秘密信息，就不可能预测结果。

计算机生成的“随机”序列都是某种被称为随机数生成器的特殊算法的结果。设计这样的算法需要一些专门的数学知识，而这些知识超出了本书的范畴。不过，我们会介绍一种实践中相当好用的随机数生成器——线性同余生成器。

指定常数 $a\geqslant 2$，$b\geqslant 1$，$x_0\geqslant 0$，而且 $a \bmod m > max(a,b,x_0)$，便可以通过使用公式

$$x_{n+1}=(ax_n+b) \bmod m$$

生成一列数字x_0，x_1，$x_2\cdots$。如果选择的a、b、m和x_0很合适，那么得到的数字序列就会显得相当随机，即便它们是通过特定算法由“种子”x_0计算得出的。

随机数生成器生成的序列有很多用途。例如，可以根据上述序列选取基诺游戏中的开奖号码，用上述序列中的每个数除以80，取余数，并加上1，得到1到80间的某个“随机”数。不断这样处理，除去重复的数字，直到选出20个数字。只要没人知道生成算法和种子，这一游戏就可视为是公平的。

现在要计数的情况是从80个数字中选出20个，其中含有玩家所选择5个数字中的3个，以及玩家没有选择的75个数字中的17个。从5个数字中选出3个共有 $\binom{5}{3}=10$ 种方式，而从剩下的75个数字中选取17个的方式共有$\binom{75}{17}=\frac{75!}{17!58!}$种。

因此，玩家在5个数字中猜中3个的结果数与总选择数的比就是

$$\frac{10\dfrac{75!}{17!58!}}{\dfrac{80!}{20!60!}}$$

如果将上下都乘上$\frac{20!60!}{80!}$，上式就成了

$$10\left(\frac{75!}{17!58!}\right)\left(\frac{20!60!}{80!}\right)$$

可以看到分子和分母中的阶乘项很接近，几乎可以约去。例如，分子中的75!和分母中的80!就可以用分母中80到76这5个数的乘积代替，所以可以简化为

$$\frac{10\times60\times59\times20\times19\times18}{80\times79\times78\times77\times76}$$

这样就可以对可控数字加以计算了，结果是0.084。也就是说，玩家在5个数字中猜中3个的概率大约为8.4%。

4.9.3　基本关系

本节要审视概率的一些重要属性。首先，如果p是任一事件的概率，那么

$$0\leqslant p\leqslant1$$

也就是说，任何事件都是由0个或更多个点组成，所以它的概率不可能为负值。而且，没有任何事件会由比整个概率空间还多的点构成，所以它的概率不会超过1。

其次，设E是某个概率空间P中的事件。那么事件E的互补事件E就是P中不属于事件E的点的集合。不难看出

$$\text{PROB}(E)+\text{PROB}(\bar{E})=1$$

或者换句话说，$\text{PROB}(\bar{E})=1-\text{PROB}(E)$。原因在于，$P$中的每个点，要么在$E$中，要么在$\bar{E}$中，不可能同时在二者之中。

4.9.4　习题

(1) 使用图4-14中展示的掷两颗公平骰子的概率空间，给出以下事件的概率。
   (a) 掷出的点数为6（即两颗骰子点数之和是6）；
   (b) 掷出的点数为10；
   (c) 掷出的点数为奇数；
   (d) 掷出的点数在5到9之间。

(2) * 计算以下事件的概率。概率空间是从普通的52张扑克牌堆中按次序取两张牌的所有情况。
   (a) 至少有一张牌是A；
   (b) 两张牌的秩相同；
   (c) 两张牌花色相同；
   (d) 两张牌的秩和花色都相同；
   (e) 两张牌要么秩相同要么花色相同；
   (f) 第一张牌的秩高于第二张牌的秩。

(3) * 将飞镖掷向墙上一英尺见方的区域，击中该方形区域中任何一点的可能性都是相同的。那么在投掷飞镖时
   (a) 离中心3英寸以内的概率是多少？
   (b) 离边缘3英寸以内的概率是多少？

请注意，在该习题中，概率空间是一个一英尺见方的无限区域，其中所有点都是等可能性的。

(4) 计算玩家在基诺5点游戏中猜中如下数字的概率。
  (a) 猜中5个数字中的4个；
  (b) 猜中全部5个数字。

(5) 编写C语言程序实现线性同余随机数生成器。绘制它生成的前100个数字最低有效位各数字出现频率的直方图。该直方图应该具有什么属性？

4.10 条件概率

在本节中，我们将指定数项公式和策略，用来考虑若干事件概率之间的关系。其中一项重要的结论就是独立实验的概念。在独立实验中，一项实验的结果不影响其他实验的结果。我们还将运用一些技巧来计算某些复杂情形下的概率。

这些结论都依赖于"条件概率"的概念。不严谨地说，如果进行一次实验，而且得知事件E已经发生，那么表示这种结果的点也有可能出现在另一事件F中。图4-15就展示了这种情况。E条件下F的概率就是E发生前提下F也发生的概率。

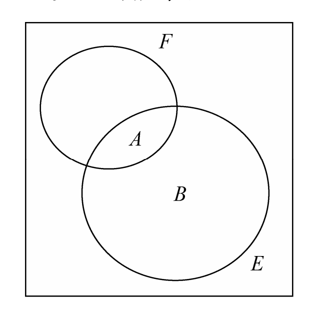


图4-15 E条件下F的概率是结果在A中的概率除以结果在A或在B中的概率

正式地讲，如果E和F是某个概率空间中的两个事件，那么E条件下F的概率，记作PROB($F|E$)，就是同时出现在E和F中的所有点的概率之和除以出现在E中各点的概率之和。在图4-15中，区域A表示那些同时在E和F中的点，B则表示在E中而不在F中的点。如果所有点都是等可能的，那么PROB($F|E$)就是A中点的数量除以A和B中点的数量之和。

✦ 示例 4.26

考虑图4-14中表示掷两颗骰子的概率空间。设事件E是第一颗骰子点数为1的6个点，F是第二颗骰子点数为1的6个点，情形如图4-16所示。既在E中又在F中的点只有一个，即点(1，1)。在E而不在F中的点共有5个。因此，条件概率PROB($F|E$)为1/6。也就是说，在确定第一颗骰子为1的条件下，第二颗骰子为1的概率是1/6。

大家可能会注意到，这一条件概率刚好等于F本身的概率。也就是说，因为F占有整个概率空间36个点中的6个点，所以PROB(F) = 6/36 = 1/6。从直观上讲，第二颗骰子掷出1的概率，并不受第一颗骰子已经掷出1这一事实的影响。我们很快就将定义"独立实验"（比如依次掷骰子）的概念，其中一次实验的结果不会对其他实验的结果产生影响。在这样的情况下，如果E和F是表示两次实验结果的事件，就可以预期PROB($F|E$) = PROB(F)。我们已经看过这一现象的一个例子了。

✦ 示例 4.27

假设实验是从有52张扑克的牌堆中按次序取两张牌。在这一不放回选择（如4.4节所述）实

验中，点的数目是$52\times51=2652$。假设这种取牌是公平的，所以每个点的可能性都是相同的。

设事件E是第一张牌为A的情况，F是第二张牌为A的情况。那么E中的点共有4×51=204个。也就是说，第一张牌是4张A中的某一张，而第二张牌可以是除去第一次选走的A之外的51张牌中的任意一张。因此，$\mathrm{PROB}(E)=204/2652$=$1/13$。这一结果是符合大家的直觉的。所有13种秩都是等可能性的，所以可以预期第一张牌出现A的可能是1/13。

同样，事件F中也有4×51=204个点。可以为第二张牌任选一种A，并在其余51张牌中任选其一作为第一张牌。第一张牌理论上讲要先取得，而这一事实是无关紧要的。因此A出现在第二张的情况共有204种。因此，与E一样，$\mathrm{PROB}(F)=1/13$。这个结果还是满足A是第二张牌的可能为1/13这一直观感受。

现在来计算$\mathrm{PROB}(F|E)$。在E的204个点中，第二张牌也为A（也就是点也在F中）的情况有12个。也就是说，E中的所有点都表示A为第一张牌的情况。选择A的方式共有4种，对应4种花色的选择。在每种选择中，第二张牌也为A的选择都有3种。因此，根据4.4节介绍的技巧，有序选择两张A的情况共有$4\times3=12$种。

因此，条件概率$\mathrm{PROB}(F|E)$是$12/204$，或者说是1/17。可以注意到，在本例中，E条件下F的概率与F的概率并不相等。这也是符合直观感受的。当第一张牌取走一张A之后，第二张牌再取到A的概率就下降了。那时候，剩下的51张牌中只有3张A，而$3/51=1/17$。与之相对的是，如果不知道第一张牌是什么，那么第二张牌就可能是52张牌中4张A中的某一张。

4.10.1　独立实验

正如示例4.23、示例4.26和示例4.27中所介绍的，有时候建立的概率空间会表示两个或多个实验的结果。在最简单的情况下，该共同概率空间中的点是结果的表，每个表代表一项实验的结果。图4-16就给出了两项实验联合起来的概率空间。在实验结果间存在联系的情形中，共同空间中可能会丢失一些点。示例4.27就讨论了这样的情况，其中共同空间表示取两张牌且结果成对，其中不可能取到两张相同的牌。

实验X独立于序列中前序实验的结果。从直观概念上讲这意味着X的各种结果都不依赖于前序实验的结果。因此，示例4.26中我们指出掷第二颗骰子是独立于掷第一颗骰子的，而在示例4.27中，我们看到取第二张牌的实验并非独立于取第一张牌的实验，因为取出第一张牌后，就不可能再次取到这张牌了。

在定义独立性时，将着重于两项实验的关系。不过，因为任一实验本身也可能是一个若干实验构成的序列，所以这样的定义有效地涵盖了具有很多实验的情况。首先必须了解表示两项成功实验X_1和X_2的结果的概率空间。

✦　示例 4.28

图4-14展示了一个共同概率空间，其中实验X_1是第一颗骰子，而实验X_2是第二颗骰子。这里每一对结果都是用一个点表示的，而这些点的可能性是相等的，都等于1/36。

在示例4.27中，我们讨论过表示按次序选取两张牌、含52个点的概率空间。该空间由(C,D)这样的牌对组成，其中C和D分别是某张扑克牌，而且$C\neq D$。这些点的可能也相同，都是1/2652。

在表示X_1接着X_2的结果的概率空间中，存在表示其中一项实验的结果的事件。也就是说，如果a是实验X_1可能出现的结果，就有一个事件是由所有表示第一项实验结果为a的点组成的。这里将该事件称为E_a。同样，如果b是实验X_2可能出现的结果，就有一个事件F_b是由所有表示第二项实验结果为b的点组成的。

✦ 示例 4.29

在图4-16中，E是E_1，就是表示第一项实验结果为1的所有点。同样，F是事件F_1，就是那些表示第二项实验结果为1的点。每一行对应着第一项实验6种可能结果中的一种，而每一列则对应第二项实验6种可能结果中的一种。

E

(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)	(1, 6)
(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)	(2, 6)
(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)	(3, 6)
(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)	(4, 6)
(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)	(5, 6)
(6, 1)	(6, 2)	(6, 3)	(6, 4)	(6, 5)	(6, 6)

F

图4-16　表示第一颗骰子或第二颗骰子点数为1的事件

严格地讲，如果对X_1的所有结果a以及X_2的所有结果b，有$\text{PROB}(F_b \mid E_a) = \text{PROB}(F_b)$，那么就说实验$X_2$是*独立于*实验$X_1$的。也就是说，不管实验$X_1$的结果是怎样的，实验$X_2$各结果的条件概率都是相同的，而且都等于实验$X_2$在整个概率空间中的概率。

✦ 示例 4.30

回到图4-16表示掷两颗骰子的概率空间，设a和b分别是1到6这些数字中的任意一个。用E_a表示第一颗骰子为a的事件，F_b表示第二颗骰子是b的事件。不难注意到，这些事件的概率均为1/6，它们各自排成一行或一列。对任意的a和b来说，$\text{PROB}(F_b \mid E_a)$也是1/6。我们在示例4.26中已经证实这一结论在$a = b = 1$的情况下是成立的，不过同样的论证过程也适用于任意两个结果a和b，因为它们的事件只有一个点是相同的。因此，掷两次骰子是相互独立的。

另一方面，在以扑克牌为例的示例4.27中，就不存在这种独立性。因为，实验X_1是第一张牌的选择，而实验X_2是从剩下的牌中选择第二张牌。考虑诸如$F_{A\spadesuit}$这样的事件，也就是说，第二张牌为黑桃A的情况。很容易便能得出该事件的概率$\text{PROB}(F_{A\spadesuit})$为1/52的结论。

再来考虑诸如$E_{3♣}$，也就是第一张牌为草花3的情况。同在$E_{3♣}$和$F_{A♠}$中的点只有一个，就是点$(3♣, A♠)$。而$E_{3♣}$中的点共有51个，也就是形如$(3♣, C)$这样的点，其中C是除了草花3之外的任意一张牌。因此，条件概率PROB($F_{A♠} | E_{3♣}$)是1/51，而不是1/52，因为这两项实验不是相互独立的。

可以考虑一个更极端的例子，就是第一张牌为黑桃A的事件$E_{A♠}$。因为$E_{A♠}$和$F_{A♠}$中没有相同的点，所以PROB($F_{A♠} | E_{A♠}$)就是0，而不是1/52。

4.10.2　概率的分配律

有时候，如果先将概率空间划分为几个区域[①]，就会让概率的计算变得更加容易。也就是说，每个点都只出现在一个区域中。通常，概率空间表示一系列实验的结果，而表示事件的区域则对应其中某一实验可能出现的结果。

假设要计算被分为R_1、R_2、…、R_k这k个区域的某个具有n个点的概率空间中事件E的概率。简单起见，假设所有点的概率都是相同的，尽管就算它们的概率不同也不影响结论的成立。设事件E由m个点组成。设区域R_i中有r_i个点（$i=1$、2、…、k）。最后，设E中处于区域R_i中的点有e_i个。请注意，$\sum_{i=1}^{k} r_i = n$，而且$\sum_{i=1}^{k} e_i = m$。原因皆在于这些点都会在某个区域中而且只会在一个区域中。

我们知道PROB(E) $= m/n$，因为m/n就是E中的点所占的部分。如果用e_i的和替代m，就得到

$$\text{PROB}(E) = \sum_{i=1}^{k} \frac{e_i}{n}$$

接着，在上述和式中每一项的分子和分母中都引入因子r_i。结果就是

$$\text{PROB}(E) = \sum_{i=1}^{k} \left(\frac{e_i}{r_i}\right)\left(\frac{r_i}{n}\right)$$

现在，请注意r_i/n就是PROB(R_i)，也就是说，r_i/n是区域R_i在整个概率空间中所占的部分。此外，e_i/r_i就是 PROB(E/R_i)，即事件R_i条件下事件E的概率。换句话说，e_i/r_i是区域R_i中也出现在E中的点的比例。结果就得到以下事件E概率的计算公式。

$$\text{PROB}(E) = \sum_{i=1}^{k} \text{PROB}\left(E \middle| R_i\right)\text{PROB}\left(R_i\right) \tag{4.10}$$

非正式地讲，E的概率是各区域的概率乘上E在相应区域中的概率的总和。

✦　示例 4.31

图4-17表示了(4.10)式的应用方式。图中展示了被垂直划分为R_1、R_2和R_3这3个区域的概率空间。其中事件E是再度用线勾勒出的。设a到f分别是所示6个组中点的数目。

设$n=a+b+c+d+e+f$。那么有PROB(R_1)$=(a+b)/n$、PROB(R_2)$=(c+d)/n$，而且PROB(R_3)$=(e+f)/n$。而事件E在这3个区域的条件概率分别是PROB($E|R_2$)$=a/(a+b)$、PROB($E|R_2$)$=c/(c+d)$，而且PROB($E|R_3$)$=e/(e+f)$。现在来评估(4.10)式，就有

① “区域”指的就是“事件”，也就是概率空间的子集。不过，这里使用术语“区域”是为了强调这是将概率空间分为完全覆盖整个区域又不互相重叠的事件。

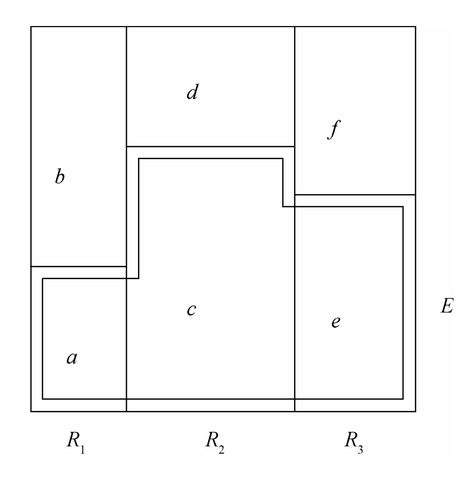


图4-17　分为区域的概率空间

$$\text{PROB}(E) = \text{PROB}(E|R_1)\text{PROB}(R_1)+\text{PROB}(E|R_2)\text{PROB}(R_2)+\text{PROB}(E|R_3)\text{PROB}(R_3)$$

如果将该式用a到f的参数表示，就是

$$\text{PROB}(E) = \left(\frac{a}{a+b}\right)\left(\frac{a+b}{n}\right)+\left(\frac{c}{c+d}\right)\left(\frac{c+d}{n}\right)+\left(\frac{e}{e+f}\right)\left(\frac{e+f}{n}\right)=\frac{a}{n}+\frac{c}{n}+\frac{e}{n}$$

请注意，直接将标记有a、c、e的3块中点的数目与整个空间的大小相比，也能得到同样的结果。$(a+c+e)/n$ 这一分数正是上式给出的E的概率，这一结果证明了(4.10)式。

✦　示例 4.32

现在利用(4.10)式计算事件E“按次序取两张扑克牌均是A”的概率。概率空间是示例4.27中讨论过的那2652个点。这里要将该空间分为R_1、R_2两个区域。

R_1：第一张牌为A的那些点。总共有$4\times 51 = 204$个这样的点，因为第一张牌为A的情况有4种，而每种情况下对应的第二张牌都有51种选择。

R_2：剩下的2448个点。

这种情况下，(4.10)式就成了

$$\text{PROB}(E) = \text{PROB}(E|R_1)\text{PROB}(R_1)+\text{PROB}(E|R_2)\text{PROB}(R_2)$$

显然$\text{PROB}(E|R_1)$，也就是R_2条件下E的条件概率，为0。如果第一张牌不为A，那么不可能拿到两张A，因此必须计算$\text{PROB}(E|R_1)\text{PROB}(R_1)$，而这个值就是$\text{PROB}(E)$。现在有$\text{PROB}(R_1) = 204/2652 = 1/13$。换句话说，第一张牌拿到A的可能性是1/13。因为总共有13种秩，所以这一概率是说得通的。

现在需要计算$\text{PROB}(E|R_1)$。如果第一张牌是A，那么剩下的51张牌中还剩3张A。因此，$\text{PROB}(E|R_1)=3/51=1/17$。因此可以得出结论：$\text{PROB}(E)=(1/17)(1/13)=1/221$。

✦　示例 4.33

现在用(4.10)式来计算事件E“掷3颗骰子，至少出现一个1点”的概率，就像示例4.16中描述过的掷骰子游戏那样。首先，我们一定要理解，那个例子中描述的“结果”的概念与概率空间中的点并不匹配。在示例4.16中，我们建立的概率空间有56种不同的“结果”，这是骰子出现1到6点的次数。例如，“一个4点、一个5点和一个6点”是一种结果，而“两个3点和一个4点”

是另一种结果。然而，这种情况下各种结果的概率并不相同。特别要说的是，3个数字都不同的概率是某个数字出现两次的概率的两倍，是3个骰子数字都相同的概率的6倍。

尽管可以使用点对应示例4.16中那种“结果”的概率空间，不过考虑按顺序掷骰子，从而构建一个包含的点概率相等的概率空间要更为自然。这样一来就有 6^3=216 种不同的结果与按次序掷3次骰子的事件对应，而每个结果的概率均为1/216。

我们可以不使用(4.10)式，而是直接计算至少有一颗骰子为1点的概率。首先，计算没有1出现的情况数。可以为3颗骰子分配2到6之中的任一数字。这样概率空间中不含1的点共有 $5^3=125$ 个，所以有1的点就有 $216-125=91$ 个。因此，$\texttt{PROB}(E)\,91/216$，大约为42%。

上述方法虽然简短，但需要使用些许“技巧”。计算这一概率的另一种方式是“强行”将概率空间分为3个区域，对应出现一个数字、两个不同数字或3个不同数字的情况。设 R_i 是具有i个不同数字的点所在的区域。可以按照下列方式计算各区域的概率。就 R_1 而言，总共有6个点，也就是3个骰子均为1到6这些点时的情况。就 R_3 而言，根据4.4节中的规则，从6个数字中选出3个不同的数字共有 $6\times5\times4=120$ 种方式。因此 R_2 中一定是具有剩下的 $216-6-120=90$ 个点。[①] 各区域的概率分别是 $\texttt{PROB}(R_1)=6/216=1/36$、$\texttt{PROB}(R_2)=90/216=5/12$、$\texttt{PROB}(R_3)=120/216=5/9$。

接着要计算条件概率。如果出现了6个数字中的3个数字，那么其中一个为1的概率是1/2。如果出现了2个数字，那么至少出现一次1的概率是1/3。如果只有一个数字出现，那么该数字为1的概率是1/6。因此$\texttt{PROB}(E|R_1)=1/6$、$\texttt{PROB}(E|R_2)=1/3$，而且$\texttt{PROB}(E|R_3)=1/2$。将这些概率都代入(4.10)式中，就得到

$$\begin{aligned}\texttt{PROB}(E)&=(1/6)(1/36)+(1/3)(5/12)+(1/2)(5/9)\\&=1/216+5/36+5/18=91/216\end{aligned}$$

当然，这一分数和直接计算的结果是吻合的。如果能理解直接计算中的那些“诀窍”，那么直接计算的方法是相当容易的。不过，将问题分为若干区域往往是一种更为可靠的保障成功的方式。

4.10.3 独立实验的乘积法则

一类常见的概率问题是求一系列独立实验的一系列结果的概率。在这种情况下，(4.10)式具有了特别简单的形式，表明这一系列结果的概率就是每一结果概率的乘积。

首先，将概率空间等分为k个区域，就会对所有的i有$\texttt{PROB}(R_i)=1/k$，因此(4.10)式可以简化为

$$\texttt{PROB}(E)=\sum_{i=1}^{k}\frac{1}{k}\,\texttt{PROB}(E|R_i)\tag{4.11}$$

看待(4.11)式的一种实用方式是将E的概率视为各区域条件下E的概率的平均值。

现在考虑表示两项独立实验X_1和X_2的结果的概率空间。可以将该空间分为k个区域，每个区域都是点的集合，这些点表示具有某个特定值的X_1的结果，这样每个区域都具有相同的概率$1/k$。

假设要计算事件E“X_1的结果为a，且X_2的结果为b”的概率，可以使用(4.11)式。如果R_i不是对应X_1的结果a的区域，那么$\texttt{PROB}(E|R_1)=0$。因此，(4.11)式中就只剩下a区域的项了。如果说该区域为R_a，就得到

① 可以直接计算该数字，用选择会出现两次的点数的6种方式，乘上选择其余骰子点数的5种方式，再乘上选择只出现一次的那个数字所在位置的 $\binom{3}{1}=3$ 种方式。就是 6×5×3=90 种方式。

$$\text{PROB}(E) = \frac{1}{k}\,\text{PROB}(E|R_a) \tag{4.12}$$

$\text{PROB}(E|R_a)$是什么？它是在X_1的结果为a的条件下，“X_1的结果为a，且X_2的结果为b”的概率。因为给定了X_1的结果为a，所以$\text{PROB}(E|R_a)$是在X_1的结果为a的条件下，X_2的结果为b的概率。又因为X_1和X_2是相互独立的，所以$\text{PROB}(E|R_a)$就是X_2的结果为b的概率。如果X_2可能有m种结果，那么$\text{PROB}(E|R_a)$就是$1/m$。那么(4.12)式就成了

$$\text{PROB}(E) = \left(\frac{1}{k}\right)\left(\frac{1}{m}\right)$$

我们可将上述推理一般化为针对任意数量的实验。想这样做，可以令实验X_1是一系列实验，并通过对独立实验总数量的归纳来证明，有着特定序列的全部结果的概率等于每个结果的概率的乘积。

利用独立性简化计算

如果知道实验是相互独立的，就有很多机会简化概率计算。乘积法则是一个例子。另一个例子就是，只要E是表示实验X_1特定结果的点集合，F是另一个独立实验X_2特定结果的点集合，那么就有$\text{PROB}(E|F) = \text{PROB}(E)$。

原则上讲，分辨两项实验是否相互独立是个复杂的任务，涉及对表示实验结果对的概率空间的检测。不过，通常可以借助该情形的物理特性，在不进行这种计算的情况下得出实验互相独立的结论。例如，在依次掷骰子时，不存在物理学上的原因令一次投掷的结果能影响其他投掷的结果，所以它们肯定是独立实验。而从牌堆中取牌则与掷骰子的情况不同。因为取出的牌是无法在随后的过程中被再次取出的，所以不用指望连续取牌是相互独立的事件。事实上，我们已在示例4.29中看到了这种独立性的缺失。

✦ 示例 4.34

电话号码后四位为1234的概率是0.0001。每一位号码的选择都是有0到9这10种可能结果的实验。其次，每一位的选择都独立于其他位的选择，因为这里进行的是4.2节中介绍的“有放回的选择”。第一位是1的概率为1/10。同样，第二位是2的概率为1/10，而其他两位的情况也是一样的。所以4位数字依次为1234的概率就是$(1/10)^4 = 0.0001$。

4.10.4 习题

(1) 使用图4-14所示的概率空间，给出如下事件对的条件概率。
  (a) 在第一颗骰子为奇数的条件下，第二颗骰子是偶数。
  (b) 在第二颗骰子至少为3的条件下，第一颗骰子是偶数。
  (c) 在第一颗骰子为4的条件下，两颗骰子点数之和至少为7。
  (d) 在两颗骰子点数之和为8的条件下，第二颗骰子是3。

(2) 将掷骰子（见示例4.16）游戏的概率空间按照示例4.33所示划分为3个区域，使用这一划分方式与4.10式计算如下概率。
  (a) 至少出现两个1点。
  (b) 3颗骰子都是1点。

(c) 刚好有一颗骰子是1点。

(3) 证明：在掷3颗骰子的游戏中，3颗骰子点数均不同的概率，是有两颗骰子出现同一点数的概率的两倍，是3颗骰子点数均相同的概率的6倍。

(4) * 通过对n的归纳证明，如果有n项实验，而且每一项实验都是相互独立的，那么任一系列结果的概率等于对应实验各项结果概率的乘积。

(5) * 证明：如果有$\text{PROB}(F|E)=\text{PROB}(F)$，则有$\text{PROB}(F|E)=\text{PROB}(E)$。并证明：如果实验$X_1$独立于实验$X_2$，那么$X_2$也独立于$X_1$。

(6) * 考虑从W和L中作出选择组成的长度为7个字母的序列的集合。可以将其视为表示7场4胜制比赛的结果的序列，其中W表示第一支队伍获得1场比赛的胜利，而L表示第二支队伍获胜。哪支队伍先取得4场胜利则赢得这场系列赛（因此，有些比赛可能从未进行过，不过我们需要将其假设结果包含在这些点中，从而获得各点概率相等的概率空间）。

(a) 在某支队伍取得第一场比赛胜利的条件下，该队在这个系列赛中取胜的概率？

(b) 在某支队伍取得前两场比赛胜利的条件下，该队在这个系列赛中取胜的概率？

(c) 在某支队伍取得前三场比赛胜利的条件下，该队在这个系列赛中取胜的概率？

(7) ** 有3个囚犯A、B和C。他们得知3人中有一个要被枪决，而且狱警知道要枪决谁。A让狱警告诉他其他两个囚犯中哪个不会被枪决。狱警告诉A，B不会被枪决。

A推理要么是他，要么是C被枪决，所以A被枪决的概率是1/2。另一方面，对A进行推理，不管谁被枪决，狱警都知道A之外的某人不会被枪决，所以他总能回答A的问题。因此，通过该问题的提问和回答都不能判定A是否会被枪决，所以A将被枪决的概率仍是1/3，就像在提出问题之前的概率那样。

那么在经过上述系列事件后，A将被枪决的真实概率是多少？*提示*：需要构建合适的概率空间，不只要表示囚犯被选中枪决的实验，而且要表示狱警有权选择回答“B”或“C”的情况下作出某种选择的实验的概率。

(8) * 假设E是处在被分为R_1、R_2、…、R_k这k个区域的空间中的事件。证明：

$$\text{PROB}(R_j|E)=\frac{\text{PROB}(R_j)\text{PROB}(E|R_j)}{\sum_{i=1}^{k}\text{PROB}(R_i)\text{PROB}(E|R_i)}$$

该公式被称为*贝叶斯定理*（Bayes' Rule）。它给出了在已知E的条件下R_j的概率的值。针对示例4.31，使用贝叶斯定理计算$\text{PROB}(R_1|E)$、$\text{PROB}(R_2|E)$和$\text{PROB}(R_3|E)$。

4.11　概率推理

概率在计算机领域的一项重要应用就是用在事件预测系统的设计中。其中一个例子就是医疗诊断系统。理想状态下，诊断过程包含执行检测或观察症状，直到检测结果或特定症状的出现与否使医生足以确定患者所患的疾病为止。然而，在实际操作中，诊断很少是确定无疑的。诊断出的只是最为可能的疾病，或者是在进行检测和观察症状的实验条件下，条件概率最大的疾病。

现在来考虑一个特别简单的例子，此例就利用了概率的诊断风格。假设已知当患者出现头痛时，他患流感的概率为50%，也就是

$$\text{PROB}(\text{流感}\mid\text{头痛})=0.5$$

在上式中，我们将“流感”解释为“患者得了流感”这一事件的名称。同样，“头痛”是“患者自称头痛”这一事件的名称。

假设还知道当患者的体温达到或超过38.9摄氏度时，该患者得流感的概率是60%。如果将“发烧”作为“患者体温至少为38.9摄氏度”这一事件的名称，就可以将这一结论写为

$$\text{PROB}(流感 \mid 发烧) = 0.6$$

现在，考虑如下诊断情形。某患者来看医生，表示自己有头痛的症状。医生为他量了体温，发现他的体温是38.9摄氏度，那么该患者患流感的概率是多少？

这种情形如图4-18所示。其中有“流感”、“头痛”和“发烧”3个事件，将该空间分为8个区域，这里分别用a到h这些字母表示这些区域。例如，c就是“患者具有头痛症状而且患了流感，但不发烧”这一事件。

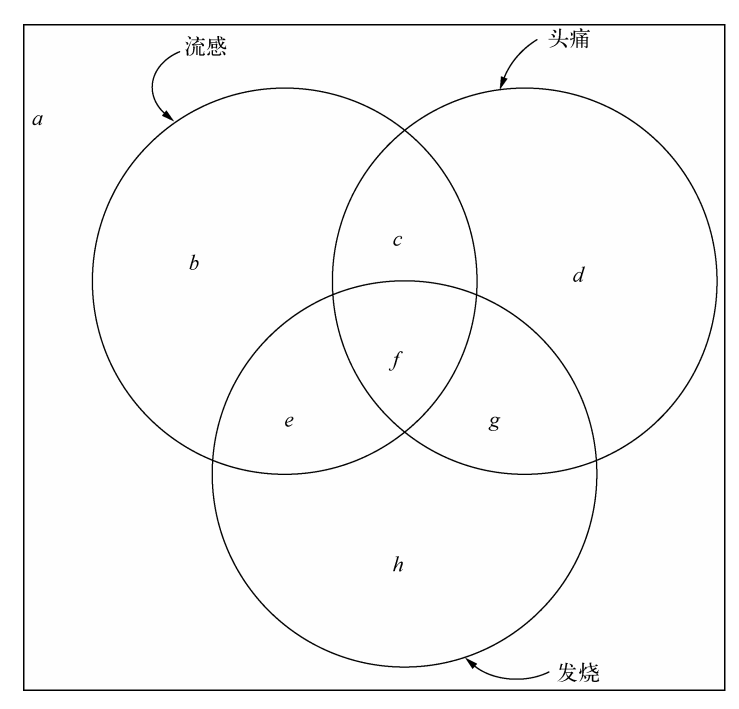


图4-18　“流感”、“头痛”和“发烧”事件

给定的这些概率信息对图4-18中事件的大小提出了一些限制。若不仅用a到h表示图4-18中的那些区域，还用这些字母表示对应事件的概率。那么条件概率PROB(流感 | 头痛) =0.5就表示区域$c+f$的和是“头痛”事件总大小的一半，或者换种形式就是

$$c + f = d + g \tag{4.13}$$

同样，PROB(流感 | 发烧) =0.5这一事实表示$e+f$是“发烧”事件总大小的3/5，或者说

$$e + f = \frac{3}{2}(g + h) \tag{4.14}$$

现在来解说“在发烧和头痛同时出现的条件下，患流感的概率是多少”这一问题。实情就是，发烧和头痛同时出现的情况表明要么是在区域f中，要么是在区域g中。在区域f中时流感的诊断是正确的，而在区域g中时则不是。因此，患流感的概率就是$f/(f+g)$。

那么$f/(f+g)$的值是多少呢？答案可能有点惊人。显然没有任何关于“流感”事件概率的信息，它可能是0，也可能是1，还可能是0和1之间的任意数字。下面有两个例子，它们分别表示图4-18所示概率空间中的点有可能出现的实际分布情况。

✦ 示例 4.35

假设图4-18中与众事件关联的概率分别是：$d = f = 0.3$，$a = h = 0.2$，而其他4个区域的概率都是0。请注意，这些值都满足(4.13)式和(4.14)式给出的限制。在本例中，$f/(f+g)=1$，也就是说，同时有头痛和发烧症状的患者肯定得了流感。那么图4-18中的概率空间实际就成了图4-19所示的样子。从该图中可看出，只要患者同时有发烧和头痛的情况，那么他肯定患了流感，而且反过来，只要他得了流感，那么肯定有发烧和头痛的症状。[①]

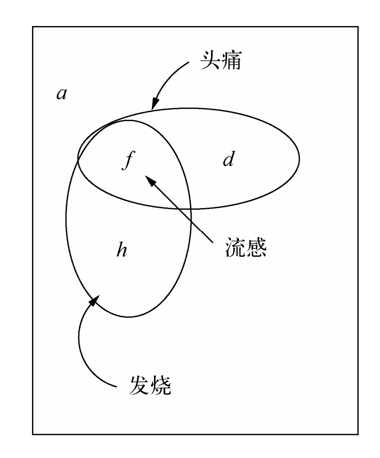


图4-19　当“发烧”和“头痛”确保“流感”时的空间示例

✦ 示例 4.36

另一个例子是给定概率$c = g = 0.2$，$a = e = 0.3$，而其他概率则是0。(4.13)式和(4.14)式还是能得到满足。不过，现在$f/(f+g)=0$。也就是说，如果患者同时有发烧和头痛的情况，那么他肯定不会得流感。这一表述相当可疑，不过(4.13)式和(4.14)式又不能推倒这一表述。这一情形如图4-20所示。

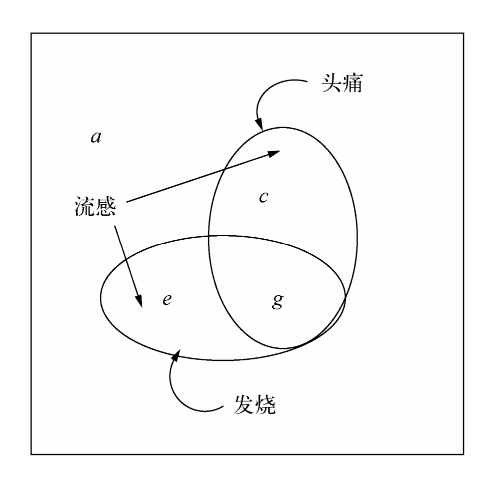


图4-20　当“发烧”和“头痛”确保不“流感”时的空间示例

① 虽然也有$b \neq 0$的其他例子，也就是患者患了流感却既不发烧也不头痛，但还是有$f/(f+g)=1$。

4.11.1 OR结合的两个事件的概率

如果没法分辨上述情形中当患者既发烧又头痛时会发生什么，我们可能想知道有没有什么可说的。在更简单的情形下，事件结合时概率的行为其实是有一些限制的。最简单的情况可能就是用析取（disjunction）或者说逻辑OR（逻辑“或”）结合两个事件时的情况了。

✦ 示例 4.37

再来看看图4-18。假设已知在任意时间，有2%的人发烧，且有3%的人感到头痛。也就是说，“发烧”事件的大小为0.02，而“头痛”事件的大小为0.03。那么，有发烧或头痛中任一种情况，或是两种情况都有的人所占比例是多大呢？

答案是至少有一种症状的人所占比例在3%到5%之间。要知道为何，用图4-18定义的8个区域来进行一些计算吧。如果“发烧”的概率为0.02，也就是说

$$e + h + f + g = 0.02 \tag{4.15}$$

如果“头痛”的概率是0.03，那么有

$$c + d + f + g = 0.03 \tag{4.16}$$

之前的问题是至少有一种症状的区域为多大，也就是问$e + h + f + g + c + d$有多大。

如果将(4.15)和(4.16)相加，就得到$e + h + 2(f + g) + c + d = 0.05$，或者换种方式表示：

$$e + h + f + g + c + d = 0.05 - (f + g) \tag{4.17}$$

因为“发烧OR头痛”的概率是(4.17)式的左边部分，所以(4.17)式的右边部分$0.05 - (f + g)$也是这一概率。

$f + g$至少为0，所以“发烧OR头痛”的概率最高是0.05，不可能超过0.05。也就是说，头痛和发烧的症状有可能从不同时出现。那么区域f和g都为空，而$e + h = 0.02$，且$c + d = 0.03$。在这种情况下，“发烧OR头痛”的概率是“发烧”的概率与“头痛”的概率之和。

那么$f + g$的最大值可能是多少？当然，$f + g$既不可能大于整个“发烧”事件，也不可能大于整个“头痛”事件。因为“发烧”更小，所以可知$f + g \leqslant 0.02$。因此，“发烧OR头痛”的最小概率可以是0.05− 0.02，也就是0.03。这一结果正好是两个事件中较大的“头痛”事件的概率，这并非巧合。换种方式看，“发烧OR头痛”概率的最小值会在两个事件中较小那个完全被较大那个包含时出现。在本例中，这种情况会在$e + h = 0$，也就是“发烧”完全包含在“头痛”中时出现。在那种情况下，除非有头痛，不然不会发烧，所以“发烧OR头痛”的概率就是“头痛”的概率——0.03。

现在可以将示例4.37中的探索一般化为针对任意两个事件，求和规则如下所示。如果E和F是任意两个事件，而G是E或F有一个发生或两者都发生的事件，那么

$$\max(\text{PROB}(E), \text{PROB}(F)) \leqslant \text{PROB}(G) \leqslant \text{PROB}(E) + \text{PROB}(F) \tag{4.18}$$

也就是说，E-OR-F的概率是在E的概率和F的概率中较大者与二者的和之间。

同样的概念放在任意其他事件H中也成立。也就是说，(4.18)中的所有概率都可以是某事件H条件下的条件概率，这样就能给出更具概括性的规则：

$$\max(\text{PROB}(E \mid H), \text{PROB}(F \mid H)) \leqslant \text{PROB}(G \mid H) \leqslant \text{PROB}(E \mid H) + \text{PROB}(F \mid H) \tag{4.19}$$

✦ 示例 4.38

假设在图4-18所示情形中已知得流感的人中有70%会发烧，而且得流感的人中有80%会头痛。那么在(4.19)中，“流感”就是事件H，E就是“发烧”事件，F是“头痛”，G是“头痛OR

发烧”。已知PROB(E | H) = PROB(发烧 | 流感) = 0.7，而PROB(F | H) = PROB(头痛 | 流感) = 0.8。

规则(4.19)表示PROB(G | H)至少是0.7和0.8中的较大者。也就是说，如果患了流感，那么发烧或头痛或两种情况都有的概率至少是0.8。规则(4.19)还表明PROB(G | H)至多是

$$\text{PROB}(E \mid H) + \text{PROB}(F \mid H)$$

或者说是 $0.7+0.8=1.5$ 。不过这一上界是没有意义的，因为事件的概率不可能大于1，所以1才是PROB(G | H)更佳的上界。

4.11.2　AND结合的事件的概率

假设已知“发烧”的概率是0.02，而“头痛”的概率是0.03。那么“发烧AND头痛”的概率是多少？也就是说，一个人同时有发烧和头痛症状的概率是多少？就像之前两个事件OR结合的情况那样，没办法给出精确的值，不过有时候可以为两个事件的合取（conjunction）或者说逻辑AND（逻辑“与”）的概率给出一些限制。

在图4-18的情况下，是要问 $f+g$ 可以有多大。我们已经得知，如果用OR关系连接事件，那么当两个事件中较小者（在该情况下是“发烧”事件）完全被另一个事件包含时， $f+g$ 会有最大值。那么，“发烧”事件的概率都集中在 $f+g$ 中，而且有 $f+g=0.02$ ，也就是“发烧”事件单独的概率。一般而言，两个事件AND结合的概率不会超过较小者的概率。

那么 $f+g$ 可以有多小？显然，没什么情况能阻止“发烧”和“头痛”完全没交集的情况出现，所以 $f+g$ 是可以为0的。也就是说，可能没人同时具有发烧和头痛的症状。

不过上述想法并不具有一般性。假设事件“发烧”和“头痛”并不是0.02和0.03这样的微小概率，而是分别有着60%和70%的概率。那么还可能说没人同时具有发烧和头痛的症状吗？如果在这种情况下还有 $f+g=0$ ，那么就有 $e+h=0.6$ ，而且 $c+d=0.7$ 。这样一来 $e+h+c+d=1.3$ ，也就是说，图4-18中的事件 $e+h+c+d$ 的概率就大于1了，而这是不可能的。

显然，两个事件的AND连接的大小不能比两事件概率之和减1还小。否则，相同的两个事件的OR连接的概率就会大于1。这一结论是在乘积法则中总结出来的。如果E和F是两个事件，而G事件是指E和F同时发生，那么

$$\text{PROB}(E)+\text{PROB}(F)-1 \leqslant \text{PROB}(G) \leqslant \min(\text{PROB}(E),\ \text{PROB}(F))$$

与求和规则一样，相同的概念也适用于另一事件H条件下的情况。也就是有

$$\text{PROB}(E \mid H)+\text{PROB}(F \mid H)-1 \leqslant \text{PROB}(G \mid H) \leqslant \min(\text{PROB}(E \mid H),\text{PROB}(F \mid H)) \tag{4.20}$$

✦　示例 4.39

再来看看图4-18。假设患流感的人中有70%会发烧，而且有80%会头痛。那么有多少人同时有发烧和头痛症状？根据(4.20)，其中H为“流感”事件，那么在某人患流感的条件下，同时有发烧和头痛症状的概率至少是 $0.7+0.8-1=0.5$ ，至多是 $\min(0.7,0.8)=0.7$ 。

涉及若干事件的规则总结

下面的内容对本节介绍的规则和4.10节中有关独立事件的规则进行了总结。假设事件E和F的概率分别为p和q，那么有下列结论。

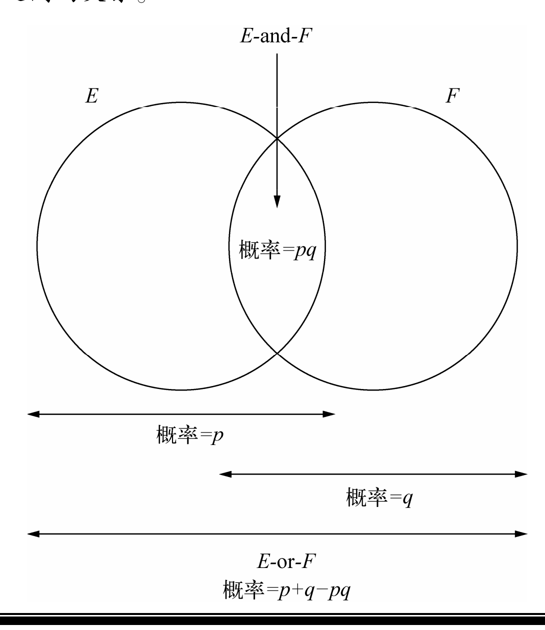
- 事件E-or-F（即E和F至少有一个发生）的概率至少是max(p,q)，且至多是p+q（或者如果p+q > 1，就是1）。

- 事件E-and-F（即E和F同时发生）的概率至多是$\min(p,q)$，且至少是$p+q-1$（或者如果$p+q < 1$，就是0）。
- 如果E和F是相互独立的事件，那么E-and-F的概率就是pq。
- 如果E和F是相互独立的事件，那么E-or-F的概率就是$p+q-pq$。

最后一个规则可能要费些思量。E-or-F的概率是$p+q$减去事件同时发生的那部分空间，因为在将E和F的概率相加时那部分空间被计算了两次。而同在E和F中的点正好是事件E-and-F，它的概率就是pq，因此，

$$\text{PROB}(E\text{-or-}F) = \text{PROB}(E) + \text{PROB}(F) - \text{PROB}(E\text{-and-}F) = p+q-pq$$

下图展示了这若干事件之间的关系。



4.11.3 处理事件间关系的一些方法

在那些需要计算复合事件（就是若干其他事件的AND或OR结果事件）的概率的应用中，往往不需要知道确切的概率。不过，我们需要确定最可能的情形或者说高概率（即概率接近1）的情形。因此，只要能推断出事件的概率为“高”，复合事件的概率范围就不太可能会带来大问题。

例如，在示例4.35引入的医疗诊断问题中，我们可能永远都没法推断出患者患流感的概率为1。不过只要结合观察到的症状和患者未出现的症状，就能得出他患流感的概率非常高，将患者诊断为流感就应该是很明智的。

然而，我们发现在示例4.35中，基本上说不出同时具有头痛和发烧症状的患者患流感的概率，即便知道每种症状都能强有力地表示患者患了流感，也是如此。真正的推理系统需要更多用来估算概率的信息或规则。作为一个简单的例子，可以明确给出PROB。流感｜头痛AND发烧。这一概率，这样就可以立刻解决该问题。

不过，如果将E_1、E_2、…、E_n这n个事件结合起来得出另一个事件F，那么就需要明确给出2^n-1个不同的概率，这些概率分别是在E_1、E_2、…、E_n中一个或多个形成的条件下F的条件概率。

✦ 示例 4.40

对 $n = 2$ 的情况，比如示例4.35的情况，就只需要给出3个条件概率。因此，正如之前所做的那样，我们可以断言PROB(流感 | 发烧) = 0.6且PROB(流感 | 头痛) = 0.5。然后，可以加上诸如PROB(流感 | 头痛AND发烧) = 0.9这样的信息。

要避免指定指数数量条件概率的情况出现，有很多种限制可帮助我们推断或估计概率。一项简单的限定就是声明某一事件表明另一事件，也就是说，第一个事件是第二个事件的子集。通常，这样的信息能提供一些有用的东西。

✦ 示例 4.41

假设我们声明只要患者患流感，他一定会头痛。那么按图4-18来看，可以说区域b和e是空的。同时假设只要患者患流感，他一定会发烧。那么图4-18中的区域c也是空的。图4-21就表示依据这两项假设简化后的图4-18。

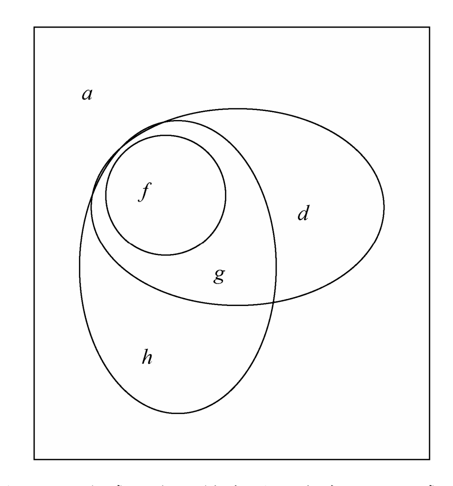


图4-21　这里，“流感”发生就表示“头痛”和“感冒”都发生

在b、c和e都为0的条件下，假设PROB(流感 | 头痛) = 0.5且PROB(流感 | 发烧) = 0.6，就可以将(4.13)式和(4.14)式改写为

$$f = d + g$$

$$f = \frac{3}{2}(g + h)$$

因为d和h都至少为0，所以第一个等式说明 $f \geqslant g$ ，而第二式说明 $f \geqslant 3g/2$ 。

再来看看同时有发烧和头痛症状的条件下患流感的概率，即PROB(流感 | 头痛AND发烧)。该条件概率在图4-18或图4-21中都是 $f/(f+g)$ 。因为 $f \geqslant 3g/2$ ，所以可得出 $f/(f+g) \geqslant 0.6$ 的结论。也就是说，同时有头痛和发烧症状的患者患流感的概率至少是0.6。

可以将示例4.41推广到任意3个事件中一个事件意味着另两个事件的情况。假设这些事件分别是E、F和G，那么

$$\text{PROB}(E|G) = \text{PROB}(F|G) = 1$$

也就是说，只要G发生，E和F肯定会发生。进一步假设PROB$(E|G) = p$，且PROB$(G|F) = q$，则有

$$\text{PROB}(G|E\text{-and-}F) \geqslant \max(p,q) \tag{4.21}$$

如果将图4-21中的“流感”、“发烧”、“头痛”分别解释为G、E、F，就可以看出(4.21)式的

推理是成立的。那么有 $p = f/(f+g+h)$ 且 $q = f/(f+g+d)$ 。因为d和h至少为0，所以可得知 $p \leqslant f/(f+g)$ 且 $q \leqslant f/(f+g)$ 。而 $f/(f+g)$ 就是PROB($G|E$-and-F)。因此，该条件概率大于等于p和q二者中的较大者。

4.11.4 习题

(1) 将求和规则和乘积法则推广到两个以上的事件上。也就是说，如果E_1、E_2、…、E_n这些事件的概率分别是p_1、p_2、…、p_n，那么回答下列问题。

(a) n个事件中至少有一件发生的概率是多少？

(b) n个事件全部发生的概率是多少？

(2) * 如果PROB($F|E$) = p，那么以下概率分别是多少？

(a) PROB($F|\bar{E}$)

(b) PROB($\bar{F}|E$)

(c) PROB($\bar{F}|\bar{E}$)

回想一下，$\bar{E}$ 是E的互补事件，而 $\bar{F}$ 是F的互补事件。

(3) 智能建筑控制会试着预测夜晚是否会“冷”，也就是说晚间温度至少比白天温度低20华氏度（约6.7摄氏度）的情况。如果控制系统知道日落前照在它传感器上的阳光指数为高，那么当晚会冷的概率就是60%，因为显然是没有云层，使得热量更容易从地面散逸。而且控制系统还知道，如果日落后一小时内温度的变化至少达到5度（约1.67摄氏度），那么晚上会冷的概率就是70%。将这3个事件分别表示为“冷”、“高”和“降”，并假设PROB(高) = 0.4且PROB(降) = 0.3。

(a) 给出PROB(高-AND-降)的上限和下限。

(b) 给出PROB(高-OR-降)的上限和下限。

(c) 假设还知道只要晚上会很冷，那么阳光传感器的读数就会高，而且日落后温度至少下降4度，即PROB(高|冷)和PROB(降|冷)都是1。给出PROB(冷|高-AND-降)的上限和下限。

(d) ** 在与(c)小题相同的假设下，给出PROB(冷|高-OR-降)的上限和下限。请注意，本题所需的推理在本节中并未提及。

(4) 在很多情况下，比如示例4.35的情况，两个或多个事件会相互强化某一结论。也就是说，我们从直觉上期望不管PROB(流感|头痛)是多少，得知患者有发烧及头痛的症状都能提高流感的概率。假设如果PROB($G|E$-AND-F) ≥ PROB($G|F$)就说事件E强化了结论G中的事件F。证明：如果事件E和F在结论G中互相强化，那么有(4.21)式成立。也就是说，E-AND-F条件下G的概率，不小于E条件下G的条件概率与F条件下G的条件概率中的较大者。

概率推理的其他应用

本节中我们已经看到了概率推理的一种重要应用：医疗推理。下面还列出了其他一些领域，在这些领域中有一些相似的概念出现在计算机解决方案中。

- 系统诊断。设备出现故障，表现出一些不正常的行为。例如，计算机屏幕一片空白，但硬盘还在运转。导致这一问题的原因是什么？
- 统筹性规划。给定经济条件的概率，比如通货膨胀以及某种商品供给的减少等，哪种战略的成功概率最大？
- 智能家电。多种高端家电可以使用概率推理（常被称为“模糊逻辑”）为用户作出决定。例如，洗衣机可以旋转并称量它盛装的衣物，预测最有可能的面料（比如免烫材料或羊毛），并据此调整洗衣的程序。

4.12　期望值的计算

通常，实验可能出现的结果都有相关联的值。在本节中，我们将利用一些简单的博彩游戏作为示例，在这些游戏中赢钱或输钱取决于实验的结果。而在下一节中，我们还将讨论计算机科学领域更复杂的示例，即计算某些算法预期运行时间的例子。

假设拥有某个概率空间，以及该空间中点上的*收益函数*f。f的*期望值*就是$f(x)\text{PROB}f(x)$上所有点x的和。用EV(f)表示该值，当所有点都是等可能时，可以通过

(1) 将空间中所有x对应的$f(x)$相加，然后

(2) 将和值除以空间中点的数目，

计算该期望值EV(f)。该期望值有时被称为*均值*，而且可以被视作“重心”。

✦　示例 4.42

假设该概率空间是表示投一颗公平骰子的结果的6个点，这些点会自然而然地被视为1到6这些整数。设该收益函数为恒等函数 $f(i)=i$, $i=1$、2、…、6，那么f的期望值就是

$$\begin{aligned}\text{EV}(f)&=\big(f(1)+f(2)+f(3)+f(4)+f(5)+f(6)\big)/6\\&=(1+2+3+4+5+6)/6=21/6=3.5\end{aligned}$$

也就是说，一颗骰子投出点数的期望值是3.5。

再看一个例子，设g是收益函数 $g(i)=i^2$ 。那么，对同样的实验——投一颗骰子，期望值g为

$$\begin{aligned}\text{EV}(g)&=(1^2+2^2+3^2+4^2+5^2+6^2)/6\\&=(1+4+9+16+25+36)/6=91/6=15.17\end{aligned}$$

非正式地讲，一颗骰子掷出点数平方的期望值是15.17。

✦　示例 4.43

再来考虑示例4.16首次引入的掷骰子游戏。该游戏的收益规则如下。玩家对某个数字下注1美元。如果该数字出现1次或多次，那么该玩家就会得到该数字出现次数那么多的美元。如果该数字未出现，那么该玩家就会输掉他下注的那些钱。

掷骰子游戏的概率空间是由1到6这几个数字的三元组构成的216个点。这些点表示掷三颗骰子的结果。我们假设玩家下注的数字是1。很明显可知，只要这些骰子都是公平的，该玩家输赢钱数的期望值就与他下注的数字无关。

该游戏的收益函数f为下列情况。

(1) $g(i,j,k)=-1$ ，如果i、j和k都不为1。也就是说，如果没出现1点，那么玩家将会输掉他下注的那1美元。

(2) $g(i,j,k)=1$ ，如果i、j或k中刚好有一个为1。

(3) $g(i,j,k)=2$ ，如果i、j或k中刚好有两个为1。

(4) $g(i,j,k)=3$ ，如果i、j和k都为1。

接下来的问题就是求g在这216个点上的平均值。因为枚举所有的点会很乏味，所以最好是先试着分别数出4种不同结果对应的点的数量。

首先，看看3颗骰子都不是1点的情况有多少种。如果没有1，则每个位置有5个数字可供选

择，所以这就成了4.2节中的分配问题。因此，没有1的情况共有 $5^3=125$ 种。按照上述的规则(1)，这125个点为收益的和值贡献了−125。

接着，数一下3颗骰子刚好有一颗为1点的情况有多少。1可以出现在3个位置中的任何一个。对每个存放1的位置，剩下两个位置都可以从5个数字中选择。因此，刚好有一个1的点共有 $3\times5\times5=75$ 个。根据规则(2)，这些点为收益贡献了+75。

而3颗骰子都为1的情况显然只有一种，所以这一概率为收益作出了+3的贡献。而剩下的216−125−72−1=15个点肯定是有两个1的，所以根据规则(2)，这些点贡献了+30的收益。

最后，将4类点对应的收益值都加起来，并除以概率空间中点的总数目，便能得出该游戏收益的期望值，因此得到

$$\mathrm{EV}(f)=(-125+75+30+3)/216=-17/216=-0.079$$

也就是说，玩家平均每下注1美元就会输掉约8美分。这一结果可能会让人吃惊，因为游戏表面上看起来是一次机会平等的打赌。这一点将在本节的习题中加以讨论。

正如示例4.43所表示的，有时候根据收益函数的值将概率空间中的点分组也更易于计算。一般而言，假设有某个收益函数为f的概率空间，而且f只产生有限数量的不同值。例如，在示例4.43中，f产生的值只有−1、1、2和3。对每个由f产生的值v，设E_v是由满足 $f(x)=v$ 的点x组成的事件。也就是说，E_v是让f产生值v的点的集合，那么

$$\mathrm{EV}(f)=\sum_v v\,\mathrm{PROB}(E_v) \tag{4.22}$$

在这些点概率相同的一般情况下，设n_v是事件E_v中点的数目，并设n是该概率空间中点的总数。那么$\mathrm{PROB}(E_v)$就是n_v/n，这样就可以有

$$\mathrm{EV}(f)=\left(\sum_v vn_v\right)/n$$

✦　示例 4.44

在示例4.25中，我们介绍了基诺游戏，并计算了在5个数字里猜中3个的概率。现在来计算一下基诺5点游戏收益的期望值。回想一下，在5点游戏中，玩家要从1到80中竞猜5个数字。在游戏开始后，会从1到80这些数字中选取20个。如果这20个数字中有3个或3个以上与玩家所选的5个数字相同，那么玩家就中奖了。

不过，收益取决于玩家所选的5个数字中猜对了多少个。通常，如果下注1美元，那么玩家所选5个数字中要是猜中3个，就可以得到2美元，也就是有1美元的净收益。如果他所选的5个数字中有4个是对的，就将得到15美元。如果5个数字全对，就能赢得300美元的奖励。如果猜中的数字不足3个，就不会得到奖励，并会输掉他投注的那1美元。

在示例4.25中，我们计算出5个数字中猜对3个的概率是0.08394（保留4位有效数字）。同样，可以计算出5个数字中猜对4个的概率是0.01209，而5个数字全对的概率是0.0006449。那么，猜对的数字不足3个的概率就是1减去这些小数，或者说约为0.90333。少于3个、对3个、对4个和对5个的收益分别为−1、+1、+14和+299。因此，利用(4.22)式就能得出基诺5点游戏的期望收益，就是

$$0.90333\times(-1)+0.08394\times1+0.01209\times14+0.0006449\times299=-0.4573$$

因此，玩家平均每在该游戏中投注1美元，就大约会损失46美分。

习题

(1) 证明：如果掷3颗骰子，出现1点的预期数量是1/2。

(2) *只要有1就能中奖，而没有1就不中，那么，为什么习题(1)中的事实并不意味着掷骰子游戏是一场机会平等的游戏（即在1或任一数字上下注的期望收益为0）？

(3) 假设在基诺4点游戏中，玩家要竞猜4个数字，而回报如下：猜中两个数字，得1美元（即玩家可以拿回他下注的那1美元）；猜中3个数字，得4美元；4个数字全中，得50美元。那么回报的期望值是多少？

(4) 假设在基诺6点游戏中回报如下：猜中3个，得1美元；猜中4个，得4美元；猜中5个，得25美元；全中，得1000美元。那么回报的期望值是多少？

(5) 假设要玩6颗骰子的掷骰子游戏。玩家会为某个数字下注1美元，然后掷出骰子。他选择的数字每出现一次，就会得到1美元的奖励。例如，如果出现一次，那么净回报为0；如果出现两次，则净回报为+1，等等。那么这是种公平游戏（即回报的期望值为0）吗？

(6) * 根据习题(5)表示的回报设计，我们可以对标准形式的掷3颗骰子的游戏的回报规则加以改变，让玩家可以下注一定数额。那么玩家下注的数字出现一次他就会得到1美元。为了使游戏成为一场公平游戏，玩家应该下注多少才合适？

4.13　概率在程序设计中的应用

在本节中，我们将考虑概率计算在计算机科学中的两类应用。第一类是对算法期望运行时间的分析。第二类则是一种常被称为“蒙特卡洛”算法的新算法，因为这种算法具有不正确的风险。而正如我们将看到的，通过对参数的调整，是有可能将蒙特卡洛算法正确的概率提高到令人满意的程度的，只不过没法让正确的概率达到1，或者说绝对正确。

4.13.1　概率分析

考虑以下简单问题。假设有一个含n个整数的数组，并询问某整数x是否为数组`A[0..n-1]`中的项。图4-22所示的算法就是完成这一工作的。请注意，它会返回`BOOLEAN`（布尔）类型，在1.6节中已经定义过它是`int`类型的，而且还定义了常量`TRUE`和`FALSE`，它们分别表示1和0。

```
    BOOLEAN find(int x, int A[], int n)
    {
        int i;

(1)     for(i = 0; i < n; i++)
(2)         if(A[i] == x)
(3)             return TRUE;
(4)     return FALSE;
    }
```

图4-22　在大小为n的数组A中找出元素x

第(1)到第(3)行会检查数组中的每一项，而且如果在数组中找到x，就立即终止循环并返回`TRUE`作为答案。而如果未找到x，则会到达第(4)行并返回`FALSE`。设循环体以及循环的递增与测试所花的时间为c。设第(4)行和循环初始化所花的时间为d。那么如果未找到x，图4-22所示函数的运行时间就是$cn+d$，也就是$O(n)$。

不过，假设找到了x，那么图4-22所示函数的运行时间又是多少？显然，越早找到x，所花的时间就越少。如果x因某种原因一定是在`A[0]`位置，那么所花的时间就是$O(1)$，因为循环只会迭代一次。不过如果x总是在末尾或接近末尾，那么所花的时间就会是$O(n)$。

当然，最坏的情况就是我们在最后一步才找到x，所以$O(n)$就是最坏情况下的平滑紧上界。不过，平均情况有没有可能比$O(n)$好得多呢？要解决这一问题，就需要定义一个概率空间，其中的点都表示x可能在的位置。最简单的假设就是x会等可能地被放置在数组A中的任意一个位置。如果这样的话，该概率空间就有n个点，每个点分别表示数组A下标的界限0到$n-1$这些整数。

接着问题又来了：在该概率空间中，图4-22所示函数运行时间的期望值是多少？考虑该空间中的点i，i可以是从0到$n-1$中的任一个。如果x是在`A[i]`的位置，循环就会迭代$i+1$次。因此运行时间的上界就是$ci+d$。不过这一界限略有常数d的偏差，因为第(4)行从未执行过。不过，这一差异是无关紧要的，因为在将运行时间转换为大O表达式时，d就会被消去了。

因此我们必须求出函数$f(i)=ci+d$在本概率空间中的期望值。将i从0到$n-1$时的$ci+d$相加，并除以点的总数量n，就得到

$$\text{EV}(f)=(\sum_{i=0}^{n-1}ci+d)/n=\big(cn(n-1)/2+dn\big)/n=c(n-1)/2+d$$

对较大的n，该表达式的值约为$cn/2$。因此，$O(n)$就是该期望值的平滑紧上界。也就是说，在大约为2的常数因子内，这个期望值与最坏的情况是相同的。这一结果从直觉上讲是成立的。如果x等可能地出现在数组中的任意位置，它"通常会"在数组的某一半中，因此大约只需要下x根本不在数列中或在最后一个元素的位置时一半的工夫即可。

4.13.2 使用概率的算法

图4-22所示的算法是确定的，它总是对同样的数据进行同样的处理。只有对期望运行时间的分析利用到了概率的计算。几乎我们遇到的每种算法都是确定的。不过，有一些问题靠虽不确定但会以某种基本方式从概率空间中进行选择的算法能更好地得到解决。从假想的概率空间中进行这样的选择并不难，方法就是利用4.9节中介绍的随机数生成器。

一类常见的概率算法是蒙特卡洛算法，在每次迭代时会进行随机选择。根据这一选择，它既有可能说"真"，就是保证会得到正确答案的情况，也有可能说"我不知道"，就是正确答案既可能为"真"也可能为"假"的情况。其可能性如图4-23中的概率空间所示。

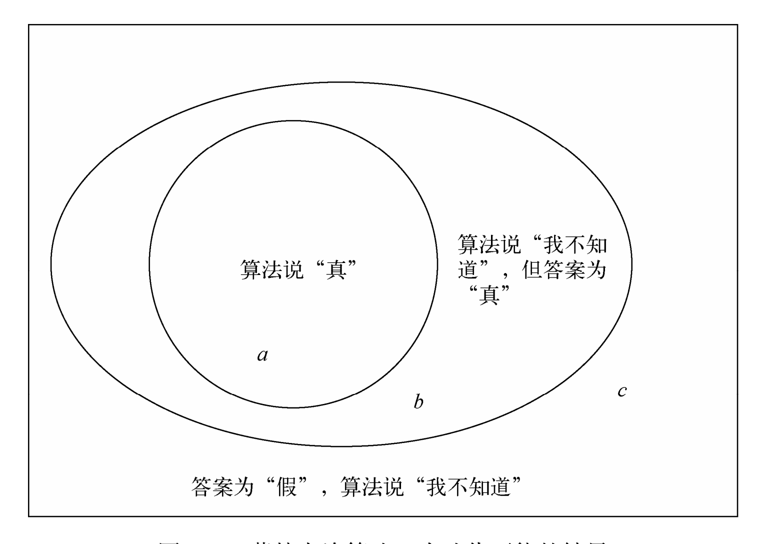


图4-23 蒙特卡洛算法一次迭代可能的结果

在答案为真的条件下，算法说“真”的概率是$a/(a+b)$。也就是说，该概率是图4-23中给定a或b的条件下事件a的条件概率。只要该概率p大于0，就可以随便迭代多少次，并迅速减小失败的概率。通过“失败”，我们表示了正确答案为“真”，但算法中没有哪次迭代可以得出这一结果。

因为每次迭代都是独立实验，如果正确答案是“真”，而且要迭代n次，那么算法从不说“真”的概率是$(1-p)^n$。只要$1-p$是严格小于1，就知道$(1-p)^n$会随着n的增长迅速减小。例如，如果$p=1/2$，那么$1-p$也是1/2。$n=10$时，$(0.5)^n$大约是1/1000（见4.2节附注栏内容），$n=20$时，这个量约是1/1 000 000，等等。n每增加10，这个量就缩小约1000倍。

蒙特卡洛算法会进行n次这样的实验。如果任意实验的答案都为“真”，那么算法的答案也为“真”。如果所有答案都为“假”，那么算法的答案为“假”。因此，

(1) 如果正确答案是“假”，该算法一定会回答“假”。

(2) 如果正确答案是“真”，该算法有$(1-p)^n$的概率回答“假”，我们可以假设这一概率非常小，因为选择了足够大的n来使它很小。该算法回答“真”的概率是$1-(1-p)^n$，这个值很可能是非常接近1的。

因此，当正确答案为“假”时是不会失败的，而当正确答案为“真”时也几乎很难失败。

✦ 示例 4.45

本例要讲一个用蒙特卡洛算法解决起来更有效的问题。XYZ计算机公司订购了若干箱芯片，这些芯片应该在出厂前都经过测试以确保都是良品。不过，XYZ公司相信某几箱芯片在出厂前未经过检测，在这种情况下任一芯片不合格的概率是1/10。XYZ公司有种简单的解决方法，就是亲自检测收到的全部芯片，不过这一过程既费钱又费时。如果一箱中有n块芯片，对该箱芯片进行测试所花的时间就是$O(n)$。

更佳的方式是利用蒙特卡洛算法。从每箱芯片中随机选出k块进行测试。如果某块芯片是坏的，就回答“真”——表示该箱芯片在出厂前未经过测试，不然这块坏芯片当时就被检出了。如果该芯片是合格的，就回答“我不知道”，并继续检测下一块芯片。如果测试的k块芯片全是良品，那么就声明整箱芯片都是良品。

就图4-23而言，区域c就表示从一箱合格芯片中选出芯片的情况；区域b是某箱芯片未经测试，但芯片凑巧合格的情况；而区域a则是某箱芯片未经测试，而且芯片不合格的情况。之前“如果某箱芯片未经测试则有1/10的芯片不合格”这一假设表示圆形区域a的面积是封闭椭圆区域a和b面积的十分之一。

现在来计算一下失败的概率即k块芯片全合格，但该箱芯片未经测试。在测试完一块芯片之后说“我不知道”的概率$1-1/10=0.9$是。因为测试每块芯片的事件都是独立的，所以对k块芯片都说“我不知道”的概率是$(0.9)^k$。假设选择$k=131$。那么失败的概率就是$(0.9)^{131}$，大约是0.000001，或者说是百万分之一。也就是说，如果某箱芯片是合格的，就永不会在该箱中找出不合格的芯片，所以我们可以笃定该箱芯片是合格的。如果某箱芯片未经测试，那么在测试的131块芯片中发现不合格芯片的概率是0.999999，而且会说该箱芯片需要全面测试。有0.000001的概率是，某箱芯片未经测试但我们还说这是一箱合格芯片，而且不需要测试该箱芯片中的其余芯片。

该算法的运行时间为$O(1)$。也就是说，测试至多131块芯片的事件是个与箱中所装芯片数n无关的常量。因此，与更直观的测试全部芯片的算法相比，测试每箱芯片的时间开销从$O(n)$降到了$O(1)$，代价是每一百万个未测试的箱子中会出错一次。

此外，通过改变在得出某箱芯片合格的结论之前所测试芯片的数量，可以让出错的概率尽可能小到让我们满意。例如，如果让测试的芯片数翻番，达到262块，那么失败的概率就成了之前的平方了，也就是成了万亿分之一，或者说是10^{-12}。还有，我们能以更高的失败率为代价节省常数倍的时间。例如，如果将测试的芯片数减半，减至每箱测试66块芯片，那么失败率就会达到约1000箱未测试芯片中就有一箱出问题。

4.13.3 习题

(1) 377、383和391哪个是质数？

(2) 假设使用图4-22所示的函数查找元素x，不过在项i处找到x的概率与$n-i$成正比。也就是说，设想一个具有$n(n+1)/2$个点的概率空间，其中n个点表示x在`A[0]`的情况，$n-1$个点表示x在`A[1]`的情况，以此类推，直到1个点表示x在`A[n-1]`的情况。该算法在本概率空间中的期望运行时间是多少？

(3) 1993年，美国职业篮球联赛（NBA）设立了由未参加季后赛的11支球队构成的选秀乐透区。战绩最差的球队拿到11张彩票，次差的球队拿到10张，以此类推，直到第11差的球队拿到1张彩票。然后随机选出一张彩票，并将第一位选秀权奖励给该彩票的所有者。那么，被选中的彩票t的所有者排名（从底部算起）的函数$f(t)$的期望值是多少？

(4) ** 继续习题(3)中描述的乐透机制。中签的球队会失去所有彩票，接着会选出代表第二位选秀权的彩票。而此次中签队伍剩下的彩票都会被收回，接着抽出第三位选秀权的彩票。那么得到第二位和第三位选秀权的队伍排名的期望值是多少？

(5) * 假设有大小为n的数组，它可能是有序的，也可能是随机装满整数。我们希望能构建某个蒙特卡洛算法，若它发现该数组是无序时就说“真”，否则就说“我不知道”。通过重复这种测试k次，我们很想知道失败的概率不超过2^{-k}。给出这样的算法。提示：确保测试是相互独立的。这里举个测试不独立的例子，我们可能测试是否有`A[0]<A[1]`，并测试`A[1]<A[2]`。这两项测试是相互独立的。然而，如果接着测试`A[0]<A[2]`，该测试就不再是独立的了，因为知道前两个关系成立的话就能肯定第三个关系是成立的。

(6) ** 假设有大小为n，且存放着从1到n这一范围内整数的数组。这些整数可能在选出时就是不同的，也可能是随机独立选出的，因此数组中可能有相等的项。给出运行时间为$O(\sqrt{n})$的蒙特卡洛算法，而且该算法随机装入的数字各不相同的概率最多只有10^{-6}。

测试整数是否为质数

尽管示例4.45不是个真正的程序，但它仍然展示了一种实用的算法原则，而且其实是衡量产品可靠性的技术的一种真实写照。一些有趣的计算机算法也用到了蒙特卡洛算法的思路。

排在首位的大概是测试某个数字是否为质数的问题。这一问题并非是无聊的数论问题。事实表明，计算机安全的诸多中心思想都涉及知道一个非常大的数为质数。粗略地讲，当使用具有n位数字的质数为信息加密时，如果要在不知道密钥的情况下解密信息，就需要从几乎所有10^n种可能中猜测。如果让n足够大，就可以确保“攻破”代码要么需要超乎寻常的运气，要么需要远超可达水平的计算时间。

因此，我们想要有种方式来测试一个非常大的数是否为质数，并希望在远小于该质数的值的时间内完成测试，理想状态下，希望测试所花的时间与数字的位数成比例，即与数字的对数成比例。检测合数（非质数）的问题似乎并不难。例如，除2之外的所有偶数都是合数，所以看起来已经解决一半问题了。同样地，那些能被3整除的数各位数字之和要能被3整除，所以可以编写一个稍慢于与数字位数存在线性关系的递归算法来测试某数能否被3整除。然而，对很多数字而言，这个问题还是很棘手。例如，377、383和391中有一个是质数，是哪一个呢？

有一种测试合数的蒙特卡洛算法。在每次迭代时，它至少有1/2的概率在被测试的数字为合数时说“真”，而且如果该数字为质数，它绝对不说“真”。下面要描述的并非确切的算法，不过除了一小部分合数外，它对大部分合数都起作用。完整的算法已经超出本书要讲的范围了。

该算法的依据是费马小定理，该定理是说，如果p为质数，且a是介于1和 $p-1$ 之间的某个整数，那么当a^{p-1}除以p时，余数为1。[①]此外，除了小部分“坏”合数之外，如果a是在1到 $p-1$ 之间的整数中随机选出的，那么a^{p-1}除以p时余数不是1的概率至少有1/2。例如，假设 $p=7$，那么1^6、2^6、…、6^6分别为1、64、729、4096、15625和46656。它们除以7的余数全是1。不过，如果 $p=6$，是个合数，那么1^5、2^5、…、2^5分别等于1、32、243、1024和3125，它们除以6的余数分别是1、2、3、4、5。只有20%是1。

因此，这种测试某数字p是否为质数的“算法”要从1到 $p-1$ 中独立且随机地选出k个整数。如果对任何选出的a来说，都有a^{p-1}/p的余数不是1，就说p为合数，否则说它是质数。如果没遇到“坏”合数，就可以说失败的概率至多为2^{-k}，因为对某个给定的a而言，合数满足测试的概率至少为1/2。如果让k为40，那么只有万亿分之一的概率将某个合数当成质数。不过，若是要处理那些“坏”合数，就需要更复杂的测试。该测试仍然是p的位数的多项式，就像上述简单测试那样。

4.14　小结

大家应该记住以下与计数相关的公式和范例。

- 将k个值分配给n个对象的方法共有k^n种。范例问题是粉刷n所房屋，其中每所房屋可从k种颜色中任选其一。
- 排列n个不同的项共有$n!$种不同方式。
- 从n项中选出k项，并为选出的k项排序，共有 $n!(n-k)!$ 种不同的方式。范例问题是为有n匹赛马参加的比赛排定冠亚季军（$k=3$）。
- 从n个对象中选出m个，不考虑顺序，有$\binom{n}{m}$或者说 $n!\big(/m!(n-m)!\big)$ 种方式。范例问题是扑克牌型问题，其中 $n=52$， $m=5$。
- 如果想排列其中存在相同项的n个项，可以按照如下方式计算排列方法数。首先有$n!$。然后，如果某个值在这n项中出现 $k>1$ 次，就除以$k!$。对每个出现超过1次的值都进行该除法处理。范例问题是计算n个字母组成的单词的构词方式，其中必须在$n!$的基础上为单词中每个出现次数 $k>1$ 的字母除以$k!$。
- 如果要将n个相同的对象放入m个容器，共有$\binom{n+m-1}{m}$种方式。范例问题是给孩子分苹果。
- 如果要将n个对象放入m个容器，而其中有一些对象是不同的，那么要依照以下方式计算分装方法数。首先有 $(n+m-1)!/(m-1)!$。然后，如果有一组有k个相同对象，而且 $k>1$，就除以$k!$。对每个出现次数超过1次的值都执行该除法处理。范例问题是将若干种水果分给孩子们。

除此之外，大家还应该记住有关概率的如下要点。

① 费马小定理的确切表述为，若p为质数，且a和p互质，则当a^{p-1}除以p时，余数恒为1。

- 概率空间由点组成，其中每个点都是某个实验的结果。每个点x都与一个被称为x的概率的非负实数相关联。某个概率空间中各点概率之和是1。
- 事件是概率空间中的点的子集，事件的概率是事件中各点概率之和。任一事件的概率都是在0到1之间的。
- 如果所有点都是等可能的，那么事件E条件下事件F的条件概率就是事件E中也在事件F中的点所占的比例。
- 如果事件F条件下事件E的条件概率与事件E本身的概率相等，就表示事件E是独立于事件F的。如果事件E是独立于事件F的，那么事件F也是独立于事件E的。
- 求和规则表明，事件E和F中有一个发生的概率，至少是两者概率中的较大者，而且不会大于两者概率之和（如果该和大于1，则是不大于1）。
- 乘积规则表明，某项实验的结果既在事件E中又在事件F中的概率不大于E和F二者概率中的较小者，并至少是两者概率之和减去1（或者如果说该值为负，则是至少为0）。
- 最后，要讲一些本章所介绍的原则在计算机科学领域的应用。
- 对于能处理具有“小于”关系的任意类型数据的排序算法，在为n个项排序时都至少需要与$n \log n$成正比的时间。
- 长度为n的位串共有2^n个。
- 随机数生成器是生成看似独立实验结果的数字序列的程序，虽然这些数字其实完全是由该程序确定的。
- 概率推理系统需要一种方式表示由若干事件形成的复合事件的概率。求和规则和乘积法则有时能帮上忙。我们还了解了其他一些为复合事件的概率设定边界的简化假设。
- 蒙特卡洛算法使用随机的数字生成期望的结果（“真”）或者完全不生成结果。通常重复该算法固定次，如果没有哪次重复过程生成答案“真”，就可以得出答案为“假”的结论，从而解决手头的问题。通过对重复的次数加以选择，可以将错误得出结果为“假”的概率调整到低得令自己满意，但不能将出错的概率降到0。

4.15　参考文献

Liu [1968]是一本传统的组合学精彩之作。Graham, Knuth, and Patashnik [1989]对这一主题进行了更深的探讨，Feller [1968]则是概率论及其应用方面的经典作品。

Rabin [1976]中介绍了测试一个数字是否为质数的蒙特卡洛算法。对该算法的讨论，以及其他一些利用到随机性的有趣的计算机安全和算法问题，可以在Dewdeney [1993]中找到。Papadimitriou [1994]中呈现了一些有关这些主题的更高深的讨论。

Dewdeney, A. K. [1993]. *The Turing Omnibus*, Computer Science Press, New York.

Feller, W. [1968]. *An Introduction to Probability Theory and Its Applications*, Third Edition, Wiley, New York.

Graham, R. L., D. E. Knuth, and O. Patashnik [1989]. *Concrete Mathematics: A Foundation for Computer Science*, Addison-Wesley, Reading, Mass.

Liu, C.-L. [1968]. *An Introduction to Combinatorial Mathematics*, McGraw-Hill, New York.

Papadimitriou, C. H. [1994]. *Computational Complexity*, Addison-Wesley, Reading, Mass.

Rabin, M. O. [1976]. “Probabilistic algorithms,” in *Algorithms and Complexity: New Directions and Recent Trends* (J. F. Traub, ed.), pp. 21–39, Academic Press, New York.

第 5 章

树

在很多情况下，信息会具有家谱或组织图中那样的分层结构或嵌套结构。为分层结构建模的抽象被称为树，而且这种数据结构是计算机科学领域中最为基础的内容之一。它是包括Lisp在内的数种程序设计语言的底层模型。

本书很多章节中介绍了不同类型的树。例如，在1.3节中，我们看到一些计算机系统中的目录和文件是如何被组织成树形结构的。2.8节中，我们利用树展示了如何递归地分割表，并在归并排序算法中将其重组。3.7节中，我们用树说明了程序中的简单语句是如何一步步组合成更为复杂的语句的。

5.1 本章主要内容

本章讨论的主要内容如下。

- 与树相关的术语和概念（5.2节）。
- 用于在程序中表示树的基础数据结构（5.3节）。
- 对树中节点进行操作的递归算法（5.4节）。
- 结构归纳法——对树进行归纳证明的方法，在这种归纳中要用小树逐渐构建成更大的树（5.5节）。
- 二叉树，树的一种变种，每个节点都只有两个子树（5.6节）。
- 二叉查找树，维护一组要进行插入和删除操作的元素的数据结构（5.7节和5.8节）。
- 优先级队列是一个可以向其中添加元素的集合，不过每次只能从中删除最大的元素。偏序树（partially ordered tree）是为了实现优先级队列而引入的一种高效数据结构，而利用被称为“堆”的平衡偏序树数据结构得到的堆排序算法，在为n个元素排序时所花的时间为$O(n\log n)$。

5.2 基本术语

树是被称为*节点*的点与被称为*边*的线的集合。一条边连接着两个不同的节点，要形成树，这一系列的节点和边必须满足某些属性，图5-1就是树的示例。

(1) 在树中，有一个节点是与众不同的，它被称为*根*。树的根通常画在其顶端。在图5-1中，根为n_1。

(2) 除根之外的每个节点c都由一条边连接到某个称为c的**父节点**的节点p。我们也将节点c称为p的**子节点**。节点的父节点要画在该节点的上方。例如，在图5-1中，n_1就是n_2、n_3和n_4的父节点，而n_2是n_5和n_6的父节点。换个角度讲，n_2、n_3和n_4都是n_1的子节点，而n_5和n_6是n_2的子节点。

(3) 如果从除根之外的任一节点n开始，移动到n的父节点，再到n的父节点的父节点，以此类推，最终到达树的根节点，就说树是**连通的**。例如，从n_7开始，移动到它的父节点n_4，然后移动到n_4的父节点，也就是根节点n_1。

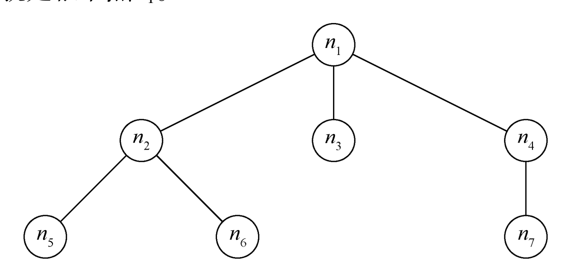


图5-1　有7个节点的树

5.2.1　树的等价递归定义

利用由较小树构成较大树的归纳定义，还可以递归地定义树。

依据。单个节点n就是一棵树，我们说n就是这棵单节点树的根。

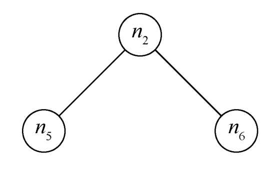
归纳。设r是一个新节点，并设T_1、T_2、…、T_k分别是根为c_1、c_2、…、c_k的树。这里要求任何节点在T_i中的出现次数都不会超过一次，而且r是个“新”节点，一定不会在这些树中出现。可以按照以下规则用r和T_1、T_2、…、T_k组成新的树T。

(a) 用r作为树T的根。

(b) 从r添加连接到c_1、c_2、…、c_k的边，使得这些节点都成为根节点r的子节点。还可以将本步骤视为让r成为T_1、T_2、…、T_k些树的根节点的父节点。

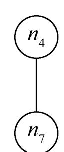
✦　示例 5.1

我们可以使用这一递归定义构建图5-1中的树，而这一构建过程也验证了图5-1中的结构为树。根据依据规则，单个节点可被视为树，所以节点n_5和n_6本身都是树。接着，可以利用归纳规则创建新树，其中n_2作为根节点r，而节点n_5就是树T_1，节点n_6则是树T_2，它们是r这一新根节点的子节点。节点c_1和c_2分别是n_5和n_6，因为它们就是树T_1和T_2的根。这样一来，我们就得出如下结构



是树的结论，而且它的根是n_2。

同样，根据依据n_7就是一棵树，而且根据归纳规则，如下结构



是一棵树，其中n_4是它的根。

节点n_3本身也是一棵树。最后，如果将节点n_1作为r，并将n_2、n_3和n_4视作刚提到的3棵树的根，就能创建出图5-1中的结构，并验证它确实是棵树。

5.2.2　路径、祖先和子孙

这种父子关系可以自然而然地扩展为祖先和子孙的关系。粗略地讲，节点的祖先就是从节点到其父节点，再到其父节点的父节点，以此类推，顺着这样一条唯一路径找到的那些节点。严格地讲，节点本身也是其自身的祖先节点。而子孙关系则是祖先关系的反向关系，像父子关系这样就是互为反向关系。也就是说，当且仅当节点a为节点d的祖先节点时，节点d才是节点a的子孙节点。

更严谨地讲，假设m_1、m_2、…、m_k是树中的一系列节点，其中m_1是m_2的父节点，m_2是m_3的父节点，以此类推，直到m_{k-1}是m_k的父节点。那么m_1、m_2、…、m_k就是该树中从m_1到m_k的一条*路径*。路径的*长度*为$k-1$，比路径上的节点数小1。请注意，路径可能是由单个节点构成的，这种情况下路径的长度就为0。

✦　示例 5.2

在图5-1中，n_1、n_2、n_6是从根节点n_1到节点n_6的一条长度为2的路径，n_1是从n_1到它自己的一条长度为0的路径。

如果m_1、m_2、…、m_k是树中的一条路径，节点m_1就是m_k的*祖先*，而节点m_k则是m_1的*子孙*。如果该路径的长度不小于1，那么m_1就是m_k的*真祖先*，而m_k则是m_1的*真子孙*。还要记住，路径长度可能为0，在这种情况下，我们就可以得出m_1是其自身的祖先也是其自身的子孙这一结论，虽然它不是自己的真祖先或真子孙。树的根节点是树中每个节点的祖先，而树中每个节点都是根节点的子孙。

✦　示例 5.3

在图5-1中，7个节点全都是n_1的子孙，而n_1是所有节点的祖先。此外，除n_1之外的所有节点都是n_1的真子孙，而n_1也是除了它自己之外的所有节点的真祖先。n_5、n_2和n_1都是n_5的祖先。而n_4的子孙有n_4和n_7。

具有相同父节点的节点有时也称为*兄弟*节点。例如，在图5-1中，n_2、n_3和n_4就互为兄弟节点，而n_5和n_6互为兄弟节点。

5.2.3　子树

在树T中，某个节点n，与其所有真子孙（若存在的话），就构成了T的*子树*。而节点n就是该子树的根节点。请注意，子树要满足3个条件才能构成树：它有根节点；子树中其他节点在该子树中都有唯一的父节点；此外，沿着该子树中任意节点的父节点回溯，最终都能达到该子树的根节点。

✦　示例 5.4

再来看图5-1，节点n_3自己就是棵子树，因为n_3除了它自己之外没有别的子孙。而节点n_2、n_5和n_6也构成了一棵子树，因为这些节点都是n_2的子孙。不过，在没有节点n_5的情况下，n_2和n_6两个节点本身是不能构成子树的。最后，图5-1中的整棵树都是它自己的子树，其中根节点为n_1。

5.2.4 叶子节点和内部节点

树中没有子节点的节点就是叶子节点。内部节点是指至少有一个子节点的节点。因此，树中的每个节点要么是叶子节点，要么是内部节点，但不可能同时是这两者。树的根节点通常是内部节点，不过，如果该树只由一个节点组成，那么该节点就既是根节点也是叶子节点。

✦ 示例 5.5

在图5-1中，n_5、n_6、n_3和n_7都是叶子节点，而n_1、n_2和n_4则是内部节点。

5.2.5 高度和深度

在树中，节点n的高度是指从n到叶子节点最长路径的长度，树的高度就是根节点的高度。而节点n的深度，或者说等级（level），就是从根节点到n的路径的长度。

✦ 示例 5.6

在图5-1中，节点n_1的高度为2，n_2的高度为1，而叶子节点n_3的高度为0。其实，任意叶子节点的高度都为0。图5-1中，树的高度为2。节点n_1的深度为0，n_2的深度为1，n_5的深度为2。

5.2.6 有序树

另外，可以为任意节点的子节点指定从左到右的顺序。例如，图5-1中n_1的子节点最左边是n_2，然后是n_3，再然后是n_4。这一从左到右的排列方式可以扩展到为树中的所有节点排列顺序。如果m和n是兄弟节点，而m在n的左边，那么m的子孙全都在n的子孙的左边。

✦ 示例 5.7

在图5-1中，根为n_2的子树中所有节点（也就是n_2、n_5和n_6）都在根为n_3和n_4的子树所有节点的左边。因此，n_2、n_3和n_6都在n_3、n_4和n_7的左边。

在树中选择任意两个互不存在祖先关系的节点x和y。由于有着“在左边”的定义，x和y中其中有一个是在另一个的左边。要分辨哪个在哪个左边，就要从x和y开始沿着路径向根节点回溯。而在某一点，可能是根节点，也可能是较低的点，这两条路径会在某个如图5-2所示的节点z相遇。从x和y到z的路径分别要经过两个不同的节点m和n，可能有 $m = x$ 且（或）$n = y$，但一定有 $m \neq n$，否则这两条路径在z以下的某个位置就融合了。

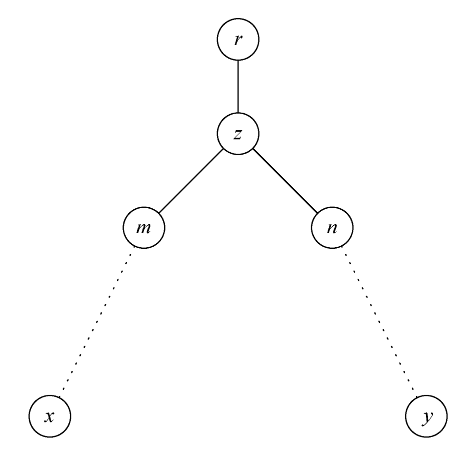


图5-2 节点x在节点y的左边

假设m在n的左边。那么因为x在根为m的子树中，而y在根为n的子树中，所以有x在y的左边。同样，如果m在n的右边，那么x也就在y的右边。

✦　示例 5.8

因为叶子节点不可能是其他叶子节点的子孙节点，所以所有叶子节点的顺序都遵循"从左起"的原则。例如，图5-1中所有叶子节点的顺序是n_5、n_6、n_3、n_7。

5.2.7　标号树

标号树（labelled tree）是指树中每个节点都有与之关联的标号或值的树。我们可将标号视为与给定节点相关联的信息。标号可以是很简单的内容，比如一个整数；也可以是复杂的内容，比如整个文档中的文本。我们可以改变节点的标号，但不能改变节点的名称。

如果节点的名称不重要，就可以用节点的标号来表示它。不过，标号并不总是能为节点提供唯一名称，因为若干个节点可能有着同一标号。因此，很多时候在绘制节点时既会标上其标号，也会标上其名称。下面的一些图展示了标号树的概念，并提供了一些示例。

5.2.8　表达式树——一类重要的树

算术表达式可以用标号树表示，而且将表达式直观表示为树往往是非常有意义的。其实，表达式树（expression tree）顾名思义就是以统一的方式指定了表达式的操作数与操作符之间的关联，不论这种关联是表达式中括号的放置需要的，还是所涉及运算符的优先级和结合规则需要的。

回想一下2.6节中对表达式的讨论，特别是示例2.17，我们在该例中给出了涉及常见算术运算符的表达式的递归定义。通过对表达式递归定义的模拟，就可以递归地定义对应的标号树。大致的概念就是通过将运算符应用到较小表达式上以构成更大的表达式，我们会创建标号为该运算符的新节点。该新节点就成为了表示较大表达式的树的根节点，而它的子节点就是表示较小表达式的树的根节点。

例如，可以按照如下方式，定义表示使用二元运算符+、−、×、/及一元运算符−的算术表达式的标号树。

依据。单个原子操作数（比如一个变量、一个整数或一个实数，如2.6节介绍的）是表达式，它的树就是标号为该操作数的一个节点。

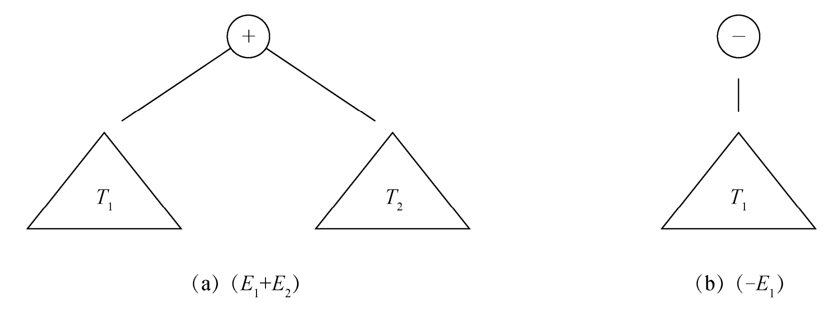


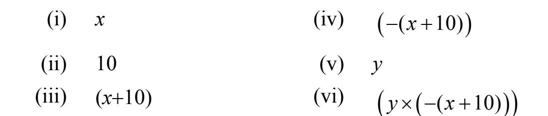
图5-3　(E_1+E_2) 和 $-E_1$ 的表达式树

归纳。如果E_1和E_2这两个表达式分别是由树T_1和T_2表示的，那么表达式(E_1+E_2)就是由图5-3a所示的树表示的，其根节点标号为+。该根节点有两个子节点，依次分别是树T_1和T_2的根节点。同样，表达式(E_1-E_2)、$(E_1\times E_2)$和(E_1/E_2)分别有着根节点标号−、×和/，且子树均为T_1和T_2的表

达式树。最后，可以将一元减号运算符应用到表达式E_1上。这里引入了标号为–的根节点，而其唯一的子节点是T_1的根节点，表示$(-E_1)$的树如图5-3b所示。

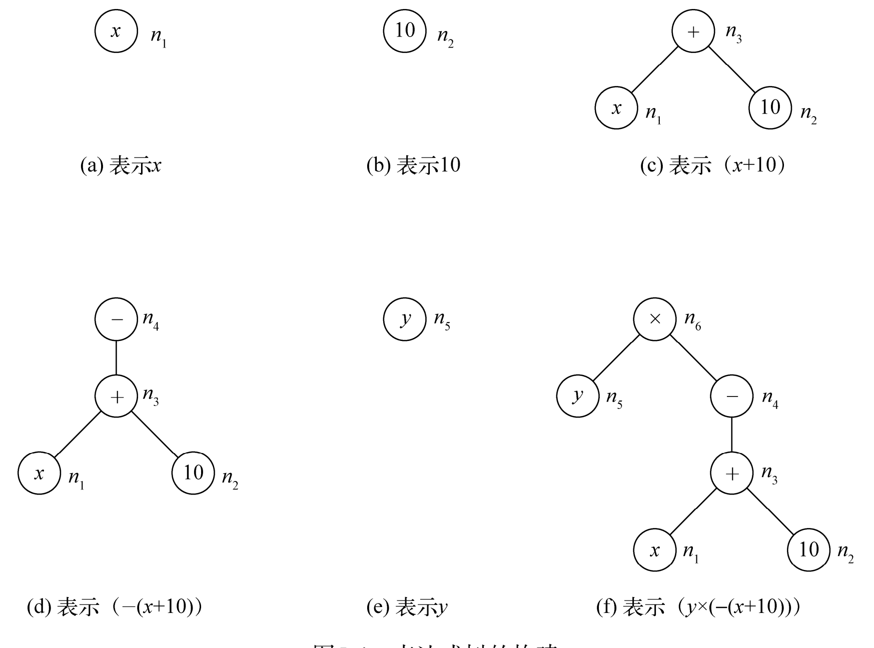
✦ 示例 5.9

在示例2.17中，我们按照依据和递归规则对一系列6个表达式的递归构建进行过讨论。图2-16列出过的这些表达式分别是：



(i)	x	(iv)	$(-(x+10))$
(ii)	10	(v)	y
(iii)	$(x+10)$	(vi)	$(y\times(-(x+10)))$

表达式(i)、(ii)和(v)都是单个操作数，因此依据规则就说明了图5-4a、图5-4b、图5-4e中的树分别是表示这些表达式的。请注意，这些树都是由一个节点组成的，节点的名称分别为n_1、n_2和n_5，而节点的标号则是圆圈中的操作数。



(a) 表示x　(b) 表示10　(c) 表示（x+10）

(d) 表示（−(x+10)）　(e) 表示y　(f) 表示（y×(−(x+10))）

图5-4 表达式树的构建

表达式(iii)是对操作数x和10应用操作符+得到的，所以可以看到图5-4c所示的表示该表达式的树根节点标号为+，而图5-4a和5-4b中树的根节点则作为其子节点。表达式(iv)是对表达式(iii)应用一元的–，所以图5-4d中表示$(-(x+10))$的树在表示$(x+10)$的树的基础之上，多了标号为–的根节点。最后，图5-4f所示的是表示表达式$(y\times(-(x+10)))$的树，其根节点标号为×，且其子节点依次为图5-4e和5-4d所示树的根节点。

5.2.9 习题

(1) 在图5-5中有一棵树，分别指出如下内容描述的各是什么。

(a) 该树的根节点。

(b) 该树的叶子节点。
(c) 该树的内部节点。
(d) 节点6的兄弟节点。
(e) 以节点5为根节点的子树。
(f) 节点10的祖先。
(g) 节点10的子孙。
(h) 节点10左边的节点。
(i) 节点10右边的节点。
(j) 该树中的最长路径。
(k) 节点3的高度。
(l) 节点13的深度。
(m) 该树的高度。

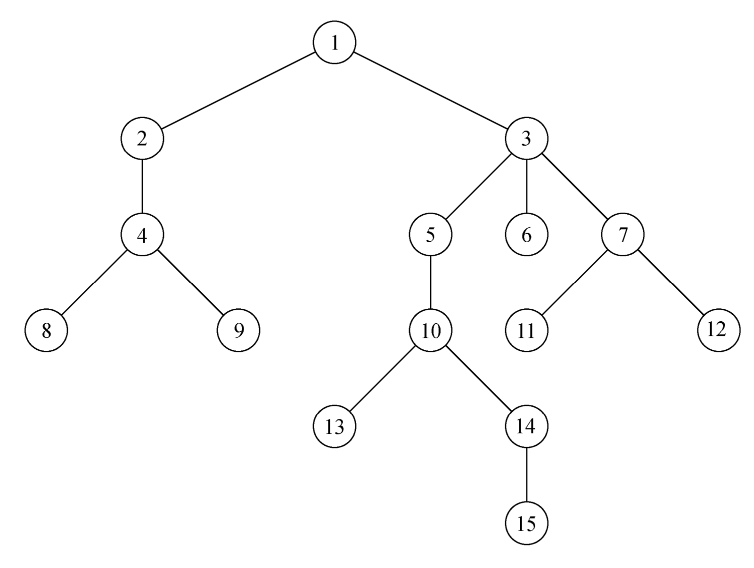


图5-5　习题(1)的树

(2) 树中的叶子节点能否有(a)子孙；(b)真子孙?

(3) 证明：在树中，任何叶子节点都不可能是其他叶子节点的祖先。

(4) *证明：本节中树的两种定义是等价的。提示：要证明由非递归定义生成的树就是根据递归定义生成的树，就要利用到对树中节点数的归纳。在相反的方向上，要利用对递归定义中递归轮数的归纳。

(5) 假设有由4个节点r、a、b和c组成的图。节点r是个孤立的点，没有边连通它。其余3个节点构成了一个循环，也就是说，有一条边连通a和b，有一条边连通b和c，还有一条边连通c和a。为何该图不是树?

(6) 在很多种树中，在内部节点和叶子节点（确切地说是这两种节点的标号）之间有着明显的区别。例如，在表达式树中，内部节点表示运算符，而叶子节点表示原子操作数。给出以下各种树的内部节点和叶子节点间的区别。

(a) 如1.3节中所述，表示目录结构的树。
(b) 如2.8节中所述，表示归并排序中表的分割和合并的树。
(c) 如3.7节中所述，表示函数的结构的树。

(7) 给出表示以下表达式的表达式树。请注意，出于习惯性的表达，题目中的表达式省略了多余的括号，大家首先必须利用运算符的优先级和结合性恢复适当的括号。

(a) $(x+1)\times(x-y+4)$

(b) $1+2+3+4+5+6$

(c) $9\times8+7\times6+5$

(8) 证明：如果x和y是有序树中两个不同的节点，那么以下条件中一定刚好有一个是成立的。

(a) x是y的真祖先。

(b) x是y的真子孙。

(c) x在y的左边。

(d) x在y的右边。

5.3 树的数据结构

很多数据结构可用来表示树。应该选用哪种数据结构，取决于想要执行的特定操作。举个简单的例子，如果想要做的只是定位节点的父节点，那么可以用由标号组成的结构体，加上指向表示节点之父节点的结构体的指针来表示每个节点。

作为一般规则，树的节点可以用结构体表示，这些结构体中的字段以某种方式将节点链接在一起，这种链接方式与节点在抽象的树中的连接方式类似，而树本身则可由指向表示根节点的结构体的指针来表示。因此，在谈到对树的表示时，我们主要是对节点的表示方式感兴趣。

表示方式的一种差异体现在表示节点的结构体在计算机内存中的位置上。在C语言中，可以使用标准库stdlib.h中的`malloc`函数为表示节点的结构体创建空间，这种情况下，节点都“漂泊”在内存中，并只能通过指针来访问。此外，可以创建由结构体组成的数组，并用数组中的元素来表示节点。这里节点还是根据它们在树中的位置链接起来的，不过也可以沿着数组的顺序来访问这些节点。因此，可以不沿着树中的路径来访问节点。基于数组的表示方式的弊端，在于没办法创建一棵节点数超过数组所含元素数量的树。接着，我们会假设这些节点都是由`malloc`创建的，虽然在树的大小有限制的情况下，由相同类型的结构体组成数组是可行的，而且有可能是首选方案。

5.3.1 树的指针数组表示

表示树的最简单方式之一就是为每个节点使用一个由表示节点标号的字段组成的结构体，后面再跟上指向该节点子节点的指针组成的数组。图5-6就表示了这样的结构。常量bf是该指针数组的大小，它表示节点可以具有的最大子节点数，这个量就是**分支系数**（branching factor）。某节点对应数组的第i个位置含有指向该节点第i个子节点的指针，不存在的子节点可用`NULL`指针表示。

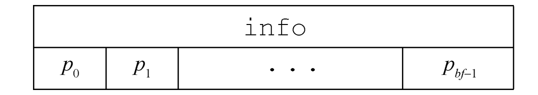


图5-6 用指针数组表示的节点

在C语言中，该数据结构可以用如下类型声明来表示。

```
typedef struct NODE *pNODE;
struct NODE {
    int info;
    pNODE children[BF];
};
```

在这里，字段info表示构成节点标号的信息，而BF则表示分支系数的常量。在本章中我们还将看到该声明的多个变种。

在这种表示树的数据结构和多数表示树的数据结构中，都将树表示为指向根节点的指针。因此，pNODE还是树的类型。其实，可以在pNODE的位置使用类型TREE，而且在5.6节开始介绍二叉树时，我们将采纳这一约定。不过，现在还是要使用pNODE这个名称代表“指向节点的指针”类型，因为在某些数据结构中，指向节点的指针除了表示树之外还用于其他用途。

指针数组表示使我们能在 $O(1)$ 的时间内访问任意节点的第i个子节点。然而，当树中只有少量节点有很多子节点时，这种表示会非常浪费空间。在这种情况下，数组中的多数指针都是NULL。

试着记住trie

术语trie（单词查找树）来源于单词retrieval（检索）的中间部分。它本来被人们读作tree，好在现在常见读法已经将其读为发音有区别的try了。

✦ 示例 5.10

树可以用来表示一系列单词，其表示方式可以使检查给定字符序列是否为存在的单词变得非常有效率。在这类称为单词查找树的树中，除了根节点之外，每个节点都有与之相关联的字母。由某个节点n表示的字符串，就是从根节点到n的路径上的字母序列。给定一组单词，单词查找树的节点就是那些表示该集合中某个单词的前缀的字符串。节点的标号是由表示该节点的字母，以及表明从根节点到该节点的字母串能否构成完整单词的布尔值组成的。如果能，就用布尔值1表示，如果不能，就用0表示。[①]

例如，假设我们的“字典”是由4个单词he、hers、his和she组成的。这些单词的单词查找树如图5-7所示。要确定单词he是否在集合中，可以从根节点n_1开始，移动到标号为h的子节点n_2，再从节点n_2移动到标号为e的子节点n_4。因为这些节点都出现在树中，而且n_4的标号中还有1，所以可以得出he在该集合中的结论。

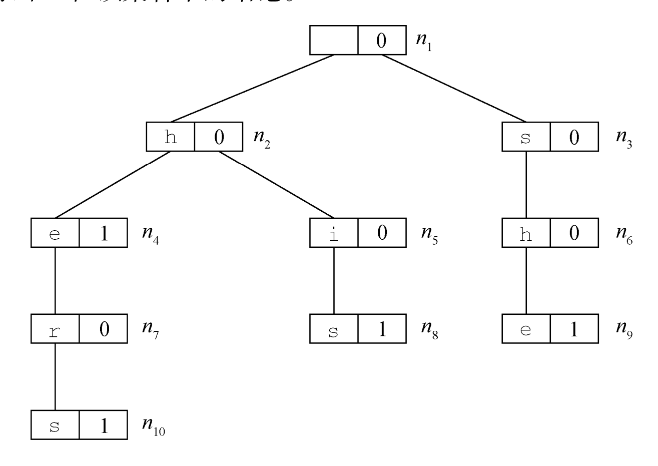


图5-7 单词he、hers、his和she的单词查找树

① 在5.2节中，介绍过的标号都只有一个值。不过，值可以是任意类型的，而且标号可以是由两个或多个字段组成的结构体。在本例中，标号有一个字段是个字母，而第二个字段则是一个值要么为0要么为1的整数。

再举一个例子，假设想要确定him是否在该集合中。可以从根节点开始沿着路径移动到n_2，再移动到n_5，这是表示前缀hi的。不过在节点n_5处找不到对应字母m的子节点。所以可以得出him不在该集合中的结论。最后，如果查找单词her，那么可以找出从根节点到节点n_7的路径。该节点存在，但标号不含1。因此可以得出her不在该集合中的结论，虽然以它为真前缀的单词hers在该集合中。

单词查找树中众节点的分支系数就等于构成这些单词的字母表中不同字符的数目。例如，如果不区分大小写字母，而且单词中不含撇号这样的特殊字符，那么分支系数就等于26。包含两个标号字段的节点的类型可以按照图5-8中所示的方式定义。在数组children中，可以假设字母a是用下标0表示的，而下标1表示字母b，以此类推。

```
typedef struct NODE *pNODE;
struct NODE {
    char letter;
    int isWord;
    pNODE children[BF];
};
```

图5-8　字母单词查找树的定义

图5-7中抽象形式的单词查找树可以用图5-9所示的数据结构表示。通过展示前两个字段letter和isWord，以及数组children中那些具有非NULL指针的元素，从而表示节点。在children数组中，对每个非NULL的元素，标记该数组的字母是由指向子节点的指针上方的项表示的，不过该字母实际上没有出现在该结构中。请注意，根节点的letter字段是无关紧要的。

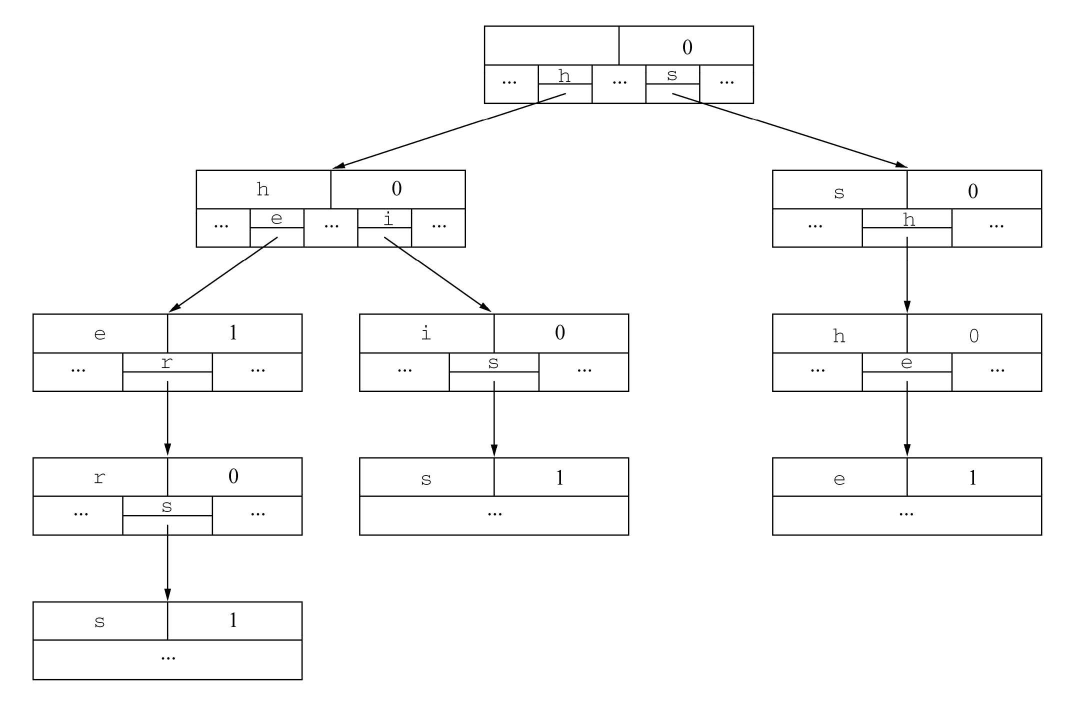


图5-9　图5-7中所示单词查找树的数据结构

5.3.2　树的最左子节点右兄弟节点表示

使用指针数组表示节点的空间利用率可能很低，因为通常情况下，绝大多数指针都会是`NULL`。图5-9显然就是这种情况，其中没有哪个节点有两个以上非`NULL`指针。事实上，如果想想这种情况，就会发现，在任何基于26个字母的字母表的单词查找树中，指针的数量都会是表示节点的指针的数量的26倍。因为没有哪个节点会有两个父节点，而且根节点是没有父节点的，所以N个节点中只有N–1个非`NULL`的指针，也就是说，每26个指针中只有不到1个是有用的。

要克服树的指针数组表示空间利用率低的问题，方法之一就是使用链表来表示节点的子节点。节点对应链表所占据的空间是与该节点子节点的数量成正比的。不过，这种表示方式在时间上要付出代价，访问第i个子节点所需时间为$O(i)$，因为在到达第i个节点之前必须遍历长度为$i-1$的链表。与之相比，使用指针数组表示子节点的话，就可以在$O(1)$时间内到达第i个子节点，跟i完全没有关系。

在树的这种*最左子节点右兄弟节点*（leftmost-child-right-sibling）表示中，要为每个节点放入一个指向其最左子节点的指针，而节点没有指向它其他子节点的指针。要找到节点n的第二个及后续的子节点，可以为这些节点创建一个链表，其中每个子节点c都指向n的子节点中紧挨在c右侧的那个，该节点称为c的*右兄弟节点*。

✦　示例 5.11

在图5-1中，n_3是n_2的右兄弟节点，n_4是n_3的右兄弟节点，n_4没有右兄弟节点。沿着n_1指向其最左子节点n_2的指针，然后移到指向n_2右兄弟节点n_3的指针，接着再到指向n_3右兄弟节点n_4的指针，就能找出n_1的子节点。接着就会发现一个为`NULL`的右兄弟节点指针，并知道n_1没有更多子节点了。

图5-10简要绘出了图5-1所示树的最左子节点右兄弟节点表示。向下的箭头是指最左子节点链接，而向右的箭头则是右兄弟节点链接。

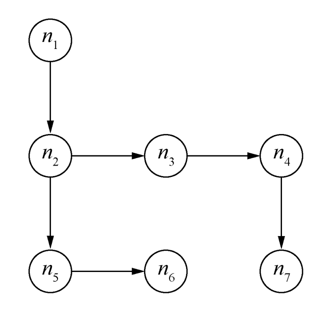


图5-10　图5-1中所示树的最左子节点右兄弟节点表示

在树的最左子节点右兄弟节点表示中，节点是按照如下方式定义的。

```
typedef struct NODE *pNODE;
struct NODE {
    int info;
    pNODE leftmostChild, rightSibling;
};
```

`info`字段存放着与节点相关联的标号，而且可以是任一类型的。字段`leftmostChild`和`rightSibling`指向最左子节点以及相应的右兄弟节点。请注意，尽管`leftmostChild`给出

了有关该节点自身的信息，但节点的rightSibling字段才是真正表示该节点父节点全部子节点的链表的那一部分。

✦ 示例 5.12

现在将图5-7中的单词查找树表示成最左子节点右兄弟节点的形式。首先，节点的类型如下。

```
typedef struct NODE *pNODE;
struct NODE {
    char letter;
    int isWord;
    pNODE leftmostChild, rightSibling;
};
```

根据示例5.10中描述过的模式，前两个字段是表示信息的。图5-7中的单词查找树可表示为图5-11所示的数据结构。请注意，每个叶子节点都有为NULL的最左子节点指针，而每个最右子节点都有为NULL的右兄弟节点指针。

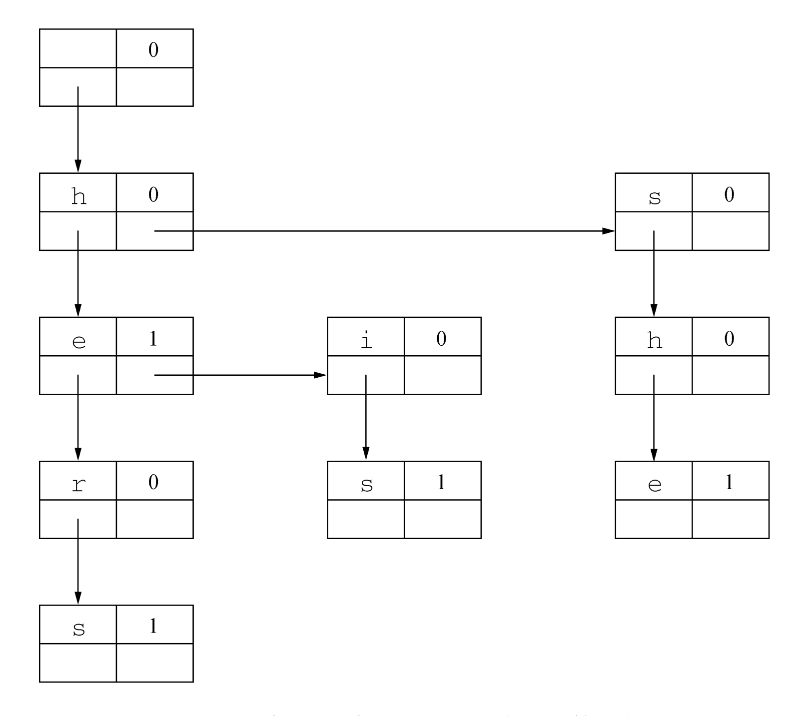


图5-11 所示单词查找树的最左子节点右兄弟节点表示

举个与最左子节点右兄弟节点表示的用途有关的例子。在图5-12中，函数seek(let, n)会接受字母*let*以及指向节点*n*的指针作为参数。它会返回一个指针，指向*n*的子节点中letter字段里有*let*的那个子节点，如果不存在这样的节点，返回的就是NULL指针。如果发现*let*，或是检查过所有的子节点，就会达到第(6)行，并跳出该循环。不管是哪种情况，c都存放着正确的值，如果存在存放了*let*的子节点，就是指向该节点的指针，如果不存在，就是NULL指针。

请注意，seek函数的运行时间与找到所要找的子节点所必须检查的子节点数成正比，如果根本找不着这样的节点，那么运行时间就与节点*n*的子节点数成正比。与之相比的是，如果使用树的指针数组表示，seek会直接返回字母*let*对应的数组元素的值，花的时间为$O(1)$。

```
    pNODE seek(char let, pNODE n)
    {
(1)     c = n->leftmostChild;
(2)     while (c != NULL)
(3)         if (c->letter == let)
(4)             break;
            else
(5)             c = c->rightSibling;
(6)     return c;
    }
```

图5-12 找到所需字母对应的子节点

5.3.3 父指针

有些时候，在表示节点结构体中包含指向其父节点的指针是很有用的，而根节点的父指针为NULL。例如，示例5.12中的结构体就成了

```
typdef struct NODE *pNODE;
struct NODE {
    char letter;
    int isWord;
    pNODE leftmostChild, rightSibling, parent;
};
```

有了这种结构体，就可以确定某给定节点表示的单词了。不断回溯父指针，直到到达根节点，我们就可以确认根节点，因为只有它的parent指针的值是NULL。这一路下来的letter字段就倒着拼出了该单词。

5.3.4 习题

(1) 对图5-5所示树中的每个节点，它们的最左子节点和右兄弟节点。

(2) 请进行下列操作。

(a) 将图5-5中的树表示为分支系数为3的单词查找树。

(b) 用最左子节点指针和右兄弟节点指针来表示图5-5中的树。

每种表示方式各需要多少字节的内存？

(3) 考虑英语中单数人称代词的如下集合：I、my、mine、me、you、your、yours、he、his、him、she、her、hers。对图5-7所示的单词查找树加以补充，从而将这13个单词都包含在内。

(4) 假设某部完整的英语词典包含了2 000 000个单词，以及1 000 000个单词前缀——也就是在其尾部加上0个或多个字母便能构成单词的字母串。

(a) 这部词典的单词查找树共有多少个节点？

(b) 假设使用示例5.10中的结构体表示节点。设指针需要4字节，且信息字段letter和isWord各需要1字节，那么这棵单词查找树需要多少字节？

(c) 在(b)小题计算出的空间中，有多少是被NULL指针占据的？

(5) 假设用示例5.12中的结构体（最左子节点右兄弟节点表示）来表示习题(4)中描述的词典。假设指针和信息字段占据的空间与习题(4)的(b)小题中的假设相同，那么这种表示中这棵树需要占据多少空间？在该空间中NULL指针占的比例又是多少？

(6) 在树中，如果节点c同为x和y的祖先，而且c的真子孙中没有一个同时是x和y的祖先，那么就说c是x和y的最低共同祖先。编写程序，使其能找出给定的树中任一对节点的最低共同祖先。在这种程序中使用什么数据结构表示树比较好？

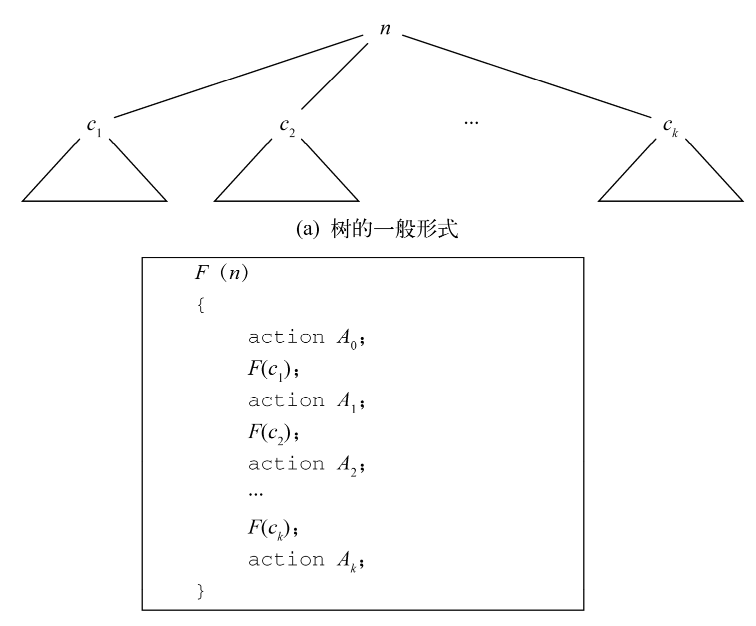
树的表示的对比

这里总结了树的指针数组表示（单词查找树）与最左子节点右兄弟节点表示的相对优势。

- 指针数组表示带来了更快的子节点访问速度，不管有多少子节点，到达任意子节点都只需要 $O(1)$ 的时间。
- 最左子节点右兄弟节点表示占用的空间更少。以图5-7所示的单词查找树为例，如果使用指针数组表示，那么每个节点含有26个指针，而如果使用最左子节点右兄弟节点表示，每个节点只含两个指针。
- 最左子节点右兄弟节点表示不要求对节点的分支系数加以限制，因此可以在不改变数据结构的前提下表示具有任一分支系数的树。然而，如果使用指针数组表示，一旦选择了数组的大小，就不能表示具有更大分支系数的树了。

5.4　对树的递归

对树进行的递归操作可以自然清晰地写下来，这样的操作数量之多突显了树的实用性。图5-13展示了接受树的节点n作为参数的递归函数 $F(n)$ 的一般形式。F首先会执行一些步骤（也可能不执行任何步骤），我们将其表示为操作A_0。接着，F会对n的第一个子节点c_1调用它自身。在这次递归调用中，F将会“探索”以c_1为根节点的子树，进行F对树进行的任何操作。当该调用返回对节点n的调用时，就会执行另一个操作A_1。接着F会在n的第二个子节点上被调用，引起对第二棵子树的探索，以此类推，就是对n的操作与在n的子节点对F的调用交替着进行。



(b) 对树进行递归的函数$F(n)$的一般形式

图5-13　对树进行递归的函数

✦ 示例 5.13

对树进行简单的递归会产生树的节点标号的前序排列（preorder listing）。这里的操作A_0是打印节点的标号，而其他的操作也无非是些“分门别类进行记录”的操作，这些操作可以让我们访问给定节点的每个子节点。效果就是，如果从根节点开始逆时针环游访问树中的每个节点，在第一次遇到这些节点时会将它们的标号打印出来。请注意，只有在第一次访问某个节点时才将其标号打印出来。这种环游如图5-14中的箭头所示，访问这些节点的顺序是$+a+*-b-c-*d*+$。这一节点标号序列的前序排列是$+a*-bcd$。

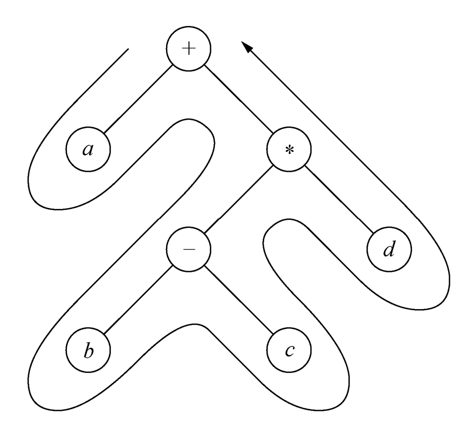


图5-14　表达式树及其环游

假设为表达式中标号为一个字母的节点使用最左子节点右兄弟节点的表示方式。内部节点的标号是该节点处的算术运算符，而叶子节点的标号是表示操作数的字母。节点和指向节点的指针可以按照如下方式定义。

```
typedef struct NODE *pNODE;
struct NODE {
    char nodeLabel;
    pNODE leftmostChild, rightSibling;
};
```

函数`preorder`如图5-15所示。在随后的解说中，可以很自然地将指向节点的指针看作节点本身。

```
        void preorder(pNODE n)
        {
            pNODE c; /* 节点n的子节点 */

(1)         printf("%c\n", n->nodeLabel);
(2)         c = n->leftmostChild;
(3)         while (c != NULL) {
(4)             preorder(c);
(5)             c = c->rightSibling;
            }
        }
```

图5-15　前序遍历函数

操作“A_0”由图5-15所示程序的以下几个部分组成。

(1) 在第(1)行，打印节点n的标号；

(2) 在第(2)行，将c初始化为n的最左子节点；

(3) 在第(3)行，执行第一次`c != NULL`的测试。

第(2)行会初始化一个循环，在该循环中，c会依次成为n的每个子节点。请注意，如果n是叶子节点，那么c就会在第(2)行被赋上`NULL`值。

第(3)行到第(5)行的`while`循环会一直进行，直到遍历完n的所有子节点。对每个子节点而言，会在第(4)行对该节点递归地调用函数`preorder`，接着在第(5)行行进到下一个子节点。$i \geqslant 1$的每个操作A_i，都是由让c在n的子节点中移动的第(5)行，以及测试是否遍历完子节点的第(3)行组成的。这些操作都只是分门别类地记录而已，与此相比，第一行中的操作A_0完成的是关键步骤：打印标号。

对图5-14中所示树的根节点调用`preorder`的一系列事件可总结为图5-16所示的情形。每一行左侧的字符就是在对`preorder(n)`的调用正在被执行时节点n的标号。因为没有哪两个节点的标号会相同，所以使用节点的标号作为其名称是没有问题的。请注意，打印出的字符是$+a*-bcd$，这一打印顺序就和环游的顺序一样。

```
      调用 preorder(+)
(+)        打印 +
(+)        调用 preorder(a)
(a)            打印 a
(+)        调用 preorder(*)
(*)            打印 *
(*)            调用 preorder(-)
(-)                打印 -
(-)                调用 preorder(b)
(b)                    打印 b
(-)                调用 preorder(c)
(c)                    打印 c
(*)            调用 preorder(d)
(d)                打印 d
```

图5-16 递归函数`preorder`对图5-14所示树进行的操作

✦ 示例 5.14

另一种为树中节点排序的常见方式是*后序*，对应图5-14所示树的环游，不过会列出最后访问的节点，而不是第一次访问的节点。例如，在图5-14中，后序排列就是$abc-d**+$。

要生成节点的后序排列，需要由最后的操作来完成打印，这样才会在对节点的所有子节点从左起依次调用后序排列函数之后，再打印该节点的标号。其他的操作则会初始化穿越子节点或移动到下一子节点的循环。请注意，如果某个节点是叶子节点，那么要做的只有列出标号，而不存在任何递归调用。

如果使用示例5.13介绍的节点表示方式，就可以通过图5-17中的递归函数`postorder`构建后序排列。在对图5-14所示树的根节点调用该函数时的操作如图5-18所示，这里使用了与图5-16中一致的节点名称转换方式。

```
     void postorder(pNODE n)
     {
         pNODE c; /* 节点n的子节点 */

(1)      c = n->leftmostChild;
(2)      while (c != NULL) {
(3)          postorder(c);
(4)          c = c->rightSibling;
         }
(5)      printf("%c\n", n->nodeLabel);
     }
```

图5-17 递归的后序函数

```
     调用preorder(+)
(+)      调用preorder(a)
(a)          打印a
(+)      调用preorder(*)
(*)          调用preorder(−)
(−)              调用 preorder(b)
(b)                  打印 b
(−)              调用 preorder(c)
(c)                  打印 c
(−)              打印 −
(*)              调用 preorder(d)
(d)              打印 d
(*)          打印 *
(+)      打印 +
```

图5-18 递归函数postorder对图5-14所示树进行的操作

✦ 示例 5.15

接下来的例子要求我们在对子树进行的所有递归调用中执行一些重大操作。假设给定一棵表达式树，其中以整数为操作数，并使用二元运算符，而且希望得出该树表示的表达式的数值。我们可以通过对该表达式树执行以下递归算法达成这一目的。

依据。对于一个叶子节点，得出该节点的值作为树的值。

归纳。假设要计算以某个节点n为根节点的子树形成的表达式的值。我们要为以n的子节点为根节点的子树所对应的子表达式求值，这两个值是节点n处的运算符对应的操作数的值。接着就可以对这两个子树的值应用标号为n的运算符，这样就得到了以n为根节点的整棵子树的值。

前缀表达式和后缀表达式

如果以前序列出表达式树的标号，就得到了给定表达式的前缀表达式。同样，以后序列出表达式树的标号就得出等价的后缀表达式。而普通概念的表达式，就是二元运算符出现在操作数之间的表达式，称为中缀表达式。例如，图5-14中表达式树的中缀表达式为$a+(b-c)*d$。正如我们在示例5.13和示例5.14中所见，等价的前缀表达式是$+a*-bcd$，等价的后缀表达式是$abc-d*+$。

有个有关前缀和后缀概念的有趣事实，只要每个运算符都有唯一的参数数量（比如，不能同时使用一元和二元的减号），那么就算没有括号，也还是能清楚地将运算符与它们对应的操作数进行分组。

可以按照如下方式由前缀表达式构建中缀表达式。在前缀表达式中，可以看到运算符后跟着所需数量的操作数，而没有内嵌的运算符。例如，在前缀表达式$+a*-bcd$中，子表达式$-bc$就是这样一个字符串，因为这个减号像此例中的所有运算符一样是二元的。我们可以用新符号来代替该子表达式，比如说设$x=-bc$，接着再重复这一确定运算符后跟其对应操作数的过程。在本例中，就是要对$+a*xb$进行处理。在这里可以确定子表达式$y=*xd$，并将剩余的字符串缩减为$+ay$。现在剩下的字符串就只有一个运算符和它的操作数了，这样就可以转换为中缀表达式$a+y$。

现在就可以通过重现这些步骤来重建中缀表达式中剩下的部分了。可以看到子表达式$y=*xd$的中缀形式为$x*d$，所以可以将$a+y$中的y替换为$x*d$，这样就得到了$a+(x*d)$。请注意，一般来说，中缀表达式里是需要括号的，虽然在本例中在为操作数分组时因为*的优先级比+高所以省略了这对括号。接着将$x=-bc$替换为中缀表达式$b-c$，便可得到最终的表达式为$a+\left((b-c)*d\right)$，这与图5-14中的树表示的表达式是相同的。

对后缀表达式来说，可以利用相似的算法。唯一的区别就是在分解后缀表达式时是要看运算符以及放在它们前面的必要数量的操作数。

我们将指向节点的指针与节点定义如下。

```
typedef struct NODE *pNODE;
struct NODE {
    char op;
    int value;
    pNODE leftmostChild, rightSibling;
};
```

字段`op`存放的要么是表示算术运算符的字符，要么是字符`i`，这里的`i`代表integer（整数），并确认节点为叶子节点。如果该节点是叶子节点，那么`value`字段就存放着该节点表示的整数，在处理内部节点时是用不上`value`的。

这一概念允许运算符具有任意数量的参数，虽然我们在编写代码时会出于简便性的考虑而假设所有运算符都是二元的。代码如图5-19所示。

如果节点n是叶子节点，第(1)行的测试会成功，并在第(2)行返回该叶子节点的整数标号。如果该节点不是叶子节点，那么会在第(3)行给它的左操作数求值，并在第(4)行给它的右操作数求值，分别将结果存入`val1`和`val2`。联系第(4)行的表示，可以注意到节点n的第二个子节点就是节点n最左子节点的右兄弟节点。第(5)行到第(9)行形成了一个`switch`语句，在该语句中要决定n处为何种运算符，并为左操作数和右操作数的值应用合适的运算。

例如，考虑图5-20中所示的表达式树。在图5-21中我们还会看到在为该表达式求值时，每个节点处进行的调用和返回的序列。和以往一样，要利用到节点标号是唯一的这一事实，并用它们的标号来为其命名。

```
     int eval(pNODE n)
     {
         int val1, val2; /* 第一棵子树和第二棵子树的值 */

(1)      if (n->op) == 'i') /* n points to a leaf */
(2)          return n->value;
         else {/* n 指向内部节点 */
(3)          val1 = eval(n->leftmostChild);
(4)          val2 = eval(n->leftmostChild->rightSibling);
(5)          switch (n->op) {
(6)              case '+': return val1 + val2;
(7)              case '-': return val1 - val2;
(8)              case '*': return val1 * val2;
(9)              case '/': return val1 / val2;
             }
         }
     }
```

图5-19 为算术表达式求值

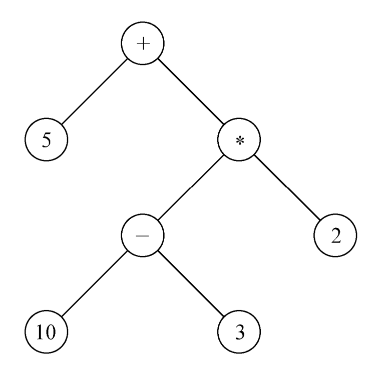


图5-20 操作数为整数的表达式树

```
    调用eval(+)
(+)     调用 eval(5)
(5)         返回 5
(+)      调用 eval(*)
(*)         调用 eval(-)
(-)             调用 eval(10)
(10)                返回 10
(-)             调用 eval(3)
(3)                 返回 3
(-)             返回 7
(*)           调用eval(2)
(2)             返回 2
(*)           返回 14
(+)        返回 19
```

图5-21 函数eval在图5-20所示树的每个节点处进行的操作

✦ 示例 5.16

有时需要确定树中各节点的高度，节点的高度可由以下函数递归地定义。

依据。叶子节点的高度为0。

归纳。内部节点的高度要比其子节点最大的高度大1。

可以将这一定义转换成递归程序，该程序会将每个节点的高度计算出来存放到`height`字段中。

依据。在叶子节点处，将高度置为0。

归纳。在内部节点，递归地计算子节点的高度，找出最大值，加上1，并将结果存储到`height`字段中。

该程序如图5-22所示，假设节点是具有如下形式的结构体。

```
typedef struct NODE *pNODE;
struct NODE {
    int height;
    pNODE leftmostChild, rightSibling;
};
```

`computeHt`函数接受指向节点的指针作为参数，并计算出该节点的高度存放到`height`字段中。如果在树的根节点处调用该函数，就会计算该树中所有节点的高度。

```
                void computeHt(pNODE n)
                {
                    pNODE c;
(1)                 n->height = 0;
(2)                 c = n->leftmostChild;
(3)                 while (c != NULL) {
(4)                     computeHt(c);
(5)                     if (c->height >= n->height)
(6)                         n->height = 1+c->height;
(7)                     c = c->rightSibling;
                    }
                }
```

图5-22　计算树中所有节点高度的例程

在第(1)行，我们会将n的高度初始化为0。如果n是叶子节点，计算就算完成了，因为第(3)行的测试将会立即失败，所以算出的任何叶子节点的高度都为0。第(2)行会将c置为（指向）n的最左子节点（的指针）。随着不断进行第(3)行至第(7)行的循环，c依次成为n的每个子节点。第(4)行会递归地计算c的高度。随着计算的进行，`n->height`中的值会比目前最高的子节点高度大1，不过如果没有子节点，这个值就是0。因此，第(5)行和第(6)行在发现比之前的子节点更高的子节点后会增加n的高度。此外，对第一个子节点，第(5)行的测试是肯定会被满足的，而且我们会将`n->height`置为比第一个子节点的高度大1。在因为处理完所有子节点而跳出循环后，`n->height`就会被置为比n子节点中的最大高度大1。

程序设计还要更具防御性

图5-19中的程序有若干方面表现出了一种粗心的编程风格，这是应该避免的。具体来说，我们在没有首先检查指针是否为NULL的情况下就一路前进了。因此，在第(1)行，n是可能为NULL的。我们真应该将程序以如下形式开头。

```
if (n != NULL) /* then do lines (1) to (9) */
else /* print an error message */
```

即便n不是NULL，在第(3)行还是可能看到它的leftmostChild字段是NULL，因此应该检测一下n->leftmostChild是否为NULL，如果是，就打印出错误消息，而且不去调用eval。同样，即便n的最左子节点存在，该子节点也可能没有右兄弟节点，所以在第(4)行之前还需要检查

```
n->leftmostChild->rightSibling != NULL
```

而且该程序还依赖于树中节点所含信息是正确的这一假设。例如，如果某个节点是内部节点，它的标号为二元运算符，而且我们已经假设它具有两个子节点，并且第(3)行和第(4)行的指针不可能为NULL。不过，运算符标号有可能是不正确的。要正确处理这种情形，就应该在switch语句中加入default情况，以检测意料之外的运算符标号。

作为一般规则，对程序的输入永远正确这一假设的依赖过分简单了；在现实中，“只要有可能出错，就肯定会出错。”如果某个程序要使用多次，势必会遇到那些形式不符合程序员预想的数据。在实践中多么小心都不为过。盲目地接受NULL指针，或假设输入数据总是正确的，都是常见的编程错误。

习题

(1) 编写递归程序，计算用最左节点指针和右兄弟节点指针表示的树的节点数量。

(2) 编写递归程序，找到树中具有最大标号的节点。假设该树节点的标号都为整数，而且是用最左子节点右兄弟节点指针表示的。

(3) 修改图5-19中的程序，使其能处理含有一元减号节点的树。

(4) * 编写递归程序，为最左子节点右兄弟节点指针表示的树计算左右对的数量。所谓左右对，就是指节点n在m左侧这样的一对节点n和m。例如，在图5-20中，节点5就在标号为*、–、10、3和2的节点左侧，而节点10在节点3和节点2左侧，节点–在节点2左侧。因此，该树的左右对共有8对。提示：在对节点n调用编写的递归函数时，要让该函数返回两个部分，以n为根节点的子树中左右对的数量，还有以n为根节点的子树中节点的数量。

(5) 以(a)前序和(b)后序列出图5-5中（见5.2节的习题）树的节点。

(6) 对如下各表达式

(i) $(x+y)*(x+z)$

(ii) $\big((x-y)*z+(y-w)\big)*x$

(iii) $\Big(\big(\big((a*x+b)*x+c\big)*x+d\big)*x+e\Big)*x+f$

完成以下操作：

(a) 构建表达式树；

(b) 写出等价的前缀表达式；

(c) 写出等价的后缀表达式。

(7) 将后缀表达式 $ab+c*de-/f$ 转换为(a)中缀表达式和(b)前缀表达式。

(8) 编写函数，使其可以“环游”树，并在经过节点时打印节点的名称。

(9) 图5-17中的后序函数进行的操作A_0、A_1等各是什么？（“操作”就是如图5-13所指的那些。）

5.5 结构归纳法

第2章和第3章已经介绍了不少有关整数属性的归纳证明。可以假设某一命题对n来说为真，

或者假设命题对所有小于等于n的整数都成立，并使用该归纳假设证明同一命题对$n+1$也成立。“结构归纳法”与之类似但不尽相同，适用于证明与树有关的属性。结构归纳法模拟了对树的递归算法，而且这种形式的归纳法在想要证明一些与树有关的命题时是最易于使用的。

假设要证明命题$S(T)$对所有的树T都为真。作为依据，要证明$S(T)$对由单一节点组成的树T为真。而对归纳部分来说，要假设T是一棵以r为根节点，并有子节点c_1、c_2、…、c_k（$k \geqslant 1$）的树。如图5-23所示，设T_1、T_2、…、T_k分别是以c_1、c_2、…、c_k为根节点的T的子树。那么归纳步骤就是假设$S(T_1)$、$S(T_2)$、…、$S(T_k)$都为真，并证明$S(T)$。如果完成了这一证明，就可以得出$S(T)$对所有的树T都成立的结论。这种形式的论证就叫作*结构归纳法*。请注意，除了要区分依据部分（1个节点）和归纳步骤（多于1个节点），结构归纳法不会提及树中具体的节点数。

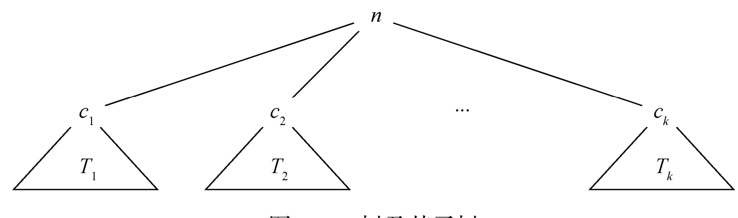


图5-23　树及其子树

✦ 示例 5.17

一般情况下，在证明对树进行操作的递归程序时会需要用到结构归纳法。举例来说，可以再次看看图5-19所示的`eval`函数，图5-24重现了该函数的函数体。只要将指向T根节点的指针赋值给该函数的参数n，就可将该函数应用于树T。然后它就会计算由T表示的表达式的值。接下来要用结构归纳法证明如下命题。

```
(1)     if (n->op) == 'i' ) /* n 指向叶子节点 */
(2)         return n->value;
        else {/* n 指向内部节点 */
(3)         val1 = eval(n->leftmostChild);
(4)         val2 = eval(n->leftmostChild->rightSibling);
(5)         switch (n->op) {
(6)             case '+': return val1 + val2;
(7)             case '-': return val1 - val2;
(8)             case '*': return val1 * val2;
(9)             case '/': return val1 / val2;
            }
        }
```

图5-24　图5-19中`eval(n)`函数的函数体

命题$S(T)$。在对T的根节点调用`eval`时，返回的值是T所表示的算术表达式的值。

依据。作为依据，T由单个节点组成。也就是说，参数n是一个（指向）叶子节点（的指针）。因为在该节点表示操作数时，`op`字段具有值“`i`”，图5-24中第(1)行的测试会成功，第(2)行会返回操作数的值。

归纳。假设节点n不是（指向）叶子节点（的指针）。归纳假设就是，$S(T')$以n的某个子节点为根节点的每棵树T'都为真。必须使用这一推理证明$S(T)$对以n为根节点的树T成立。

因为假设运算符都是二元的，所以n有两棵子树。根据归纳假设，第(3)行和第(4)行计算出

val1和val2的值，分别是左子树和右子树的值。图5-25展示了这两棵子树，val1存放着T_1的值，val2存放着T_2的值。

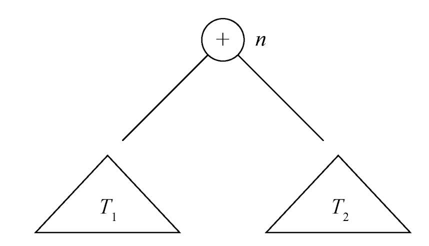


图5-25　调用*eval*(*n*)返回T_1和T_2的值的和

如果看看第(5)行到第(9)行的switch语句，就会发现不管根节点*n*处出现什么运算符，它都会被应用到val1和val2这两个值上。例如，如果根节点处存放着+，如图5-25所示，那么第(5)行返回的值就是val1+val2，正如这应该是树T_1和T_2对应表达式的和。现在就完成了归纳步骤。

因此可以得出$S(T)$对所有的表达式树*T*都成立的结论，eval函数能正确地求出表示表达式的树的值。

✦ 示例 5.18

现在来考虑一下图5-22中的computeHt函数，图5-26重现了该函数的函数体。该函数接受（指向）节点*n*（的指针）作为参数，并计算*n*的高度。我们将通过结构归纳法证明以下命题。

```
(1)            n->height = 0;
(2)            c = n->leftmostChild;
(3)            while (c != NULL) {
(4)                computeHt(c);
(5)                if (c->height >= n->height)
(6)                    n->height = 1+c->height;
(7)                c = c->rightSibling;
               }
```

图5-26　图5-22中computeHt(n)函数的函数体

命题 $S(T)$。在对指向树*T*根节点的指针调用computeHt时，*T*中每个节点的正确高度都会被存储在该节点的height字段中。

依据。如果树*T*只有一个节点*n*，那么在图5-26中的第(2)行，*c*会被赋上NULL值，因为*n*没有子节点。因此，第(3)行的测试会立即失败，而且while循环的循环体永远都不会执行。因为第(1)行会将n->height置为0（这对叶子节点而言是正确的值），所以可以得出结论，当*T*只有一个节点时$S(T)$成立。

归纳。现在假设*n*是有多个节点的树*T*的根节点，那么*n*至少有一个子节点。我们可以在归纳假设中假定，在第(4)行调用computeHt(c)时，以*c*为根节点的子树中每个节点（包括*c*本身）的height字段中都被装入了正确的高度。现在需要证明，第(3)行到第(7)行的while循环可以正确地将n->height置为比*n*的子节点的最大高度大1。要做到这一点，就需要执行另一次归纳，这是嵌套在该结构归纳法证明过程中的，就像是程序中循环嵌套在另一个循环中那样。该归纳使用的是“普通的”归纳法而不是结构归纳法，它的命题如下。

命题 $S'(i)$。在第(3)到第(7)行的循环执行i次之后，`n->height`的值要比n前i个子节点的最大高度大1。

依据。依据是$i=1$的情况。因为`n->height`在循环外——第(1)行——会被置为0，并且肯定不会有比0还小的高度，所以第(5)行的测试会得到满足。第(6)行就会将`n->height`置为比n的第一个子节点的高度大1。

归纳。假设$S'(i)$为真。也就是说，在循环的迭代i次之后，`n->height`会比前i个子节点的最大高度大1。如果有第$i+1$个子节点，那么第(3)行的测试会成功，而且会第$i+1$次执行循环体。第(5)行的测试会将新的高度与之前的最大高度相比较。如果新高度`c->height`比前i个高度中最大值加1要小，就不用对`n->height`进行任何改变。这是对的，因为前$i+1$个子节点的最大高度是可以与前i个子节点的最大高度相同的。不过，如果新的高度比之前的最大值大，第(5)行的测试就会成功，这样`n->height`就会被置为比第$i+1$个子节点的高度大1，这是正确的。

现在可以回到结构归纳了。当第(3)行的测试失败时，就已经考虑过n的所有子节点了。内层的归纳$S'(i)$说明，当子节点的总数为i时，`n->height`要比n各子节点的最大高度大1。这就是n的正确高度。将归纳假设S应用于n的各子节点，就得到结论：正确的高度已经存储到每个子节点的`height`字段中。因为已经得知n的高度也已经正确地计算出来，所以可以得出结论：T中所有节点都被赋上了它们正确的高度值。

现在已经完成了结构归纳法的归纳步骤，并得出结论：对每棵树调用`computeHt`都可以正确地计算树中每个节点的高度。

结构归纳法的模板

下面简述了进行正确的结构归纳法证明的过程。

(1) 指定要证明的命题$S(T)$，其中T是一棵树。

(2) 证明依据，也就是只要T是一棵单节点的树，$S(T)$就为真。

(3) 建立归纳步骤，设T是以r为根节点的树，而且有$k\geqslant 1$棵子树，分别为T_1、T_2、…、T_k。表示假定有归纳假设，即$S(T_i)$对每棵子树T_i（$i=1$、2、…、k）都为真。

(4) 证明在(3)中提到的假设下$S(T)$为真。

5.5.1　结构归纳法为何有效

要说明结构归纳法为何是一种有效的证明方法，其原因与普通归纳法有效的原因类似：如果结论为假，那么就会有个最小的反例，而且该反例既有可能违背依据，也可能违背归纳过程。也就是说，假设有命题$S(T)$，已经证明了它的依据和结构归纳步骤，而存在一棵树或多棵树可以让S为假。设T_0是可以让$S(T_0)$为假的这样一棵树，并设T_0与让S为假的任一棵树的节点一样少。

有两种情况。第一种，假设T_0由单个节点组成。那么根据依据，有$S(T_0)$为真，所以这种情况不可能发生。

现在就只剩下T_0有多个节点的情况了，这里假设T_0有m个节点，那么T_0就是由根节点r与一个或多个子节点构成。设以这些子节点为根的树分别是T_1、T_2、…、T_k。这里可以声明T_1、T_2、…、T_k的节点数均不超过$m-1$。因为如果某棵树（假如是T_i）的节点数达到或超过m，那么由T_i（可

能还有其他子树）和根节点r组成的T_0就至少会有$m+1$个节点。这与T_0刚好有m个节点的假设是矛盾的。

因为T_1、T_2、…、T_k这些子树的节点数都不超过$m-1$，所以就可以知道这些树都不能违背S，因为选择了T_0是让S为假的最小子树。因此可知$S(T_1)$、$S(T_2)$、…、$S(T_k)$都为真。而假设已经得到证明的归纳步骤说明了$S(T_0)$也为真，这样又与T_0违背S的假设产生了矛盾。

在考虑完这两种可能的情况后，我们就知道：不管一棵树只有一个节点还是有多个节点，T_0都不可能是S的例外。因为S是没有例外的，所以$S(T)$一定对所有的树T都为真。

5.5.2　习题

(1) 通过结构归纳法证明：

(a) 图5-15的前序遍历函数会以前序打印出树的标号；

(b) 图5-17中的后序函数会以后序列出标号。

(2) * 假设分支系数为b的单词查找树是用具有图5-6中所示格式的节点表示的。用结构归纳法证明：如果树T有n个节点，那么它的节点中有$1+(b-1)n$个NULL指针。那么，共有多少非NULL指针？

(3) * 节点的度是指节点所具有的子节点的数目。[①]用结构归纳法证明：在任意树T中，节点的数目都要比节点的度之和大1。

(4) * 用结构归纳法证明：在任意树T中，叶子节点的数目都要比具有右兄弟节点的节点的数目大1。

(5) * 用结构归纳法证明：在任意用最左子节点右兄弟节点的数据结构表示的树T中，NULL指针的数目都要比节点的数目大1。

(6) * 在5.2节开始的部分，我们给出了树的递归定义和非递归定义。使用结构归纳法证明：每棵以递归方式定义的树在非递归定义下也是同样的树。

(7) ** 证明习题(6)的逆命题：每棵以非递归方式定义的树在以递归方式定义时也是相同的树。

树的归纳的谬误形式

我们常常会想着对树的节点数进行归纳，就是假设命题对具有n个节点的树成立，并证明它对具有$n+1$个节点的树也成立。如果不够谨慎，就很可能作出这种荒谬的证明。

在第2章中对整数进行归纳证明时，我们提出了一种合理的方法，就是试着用$S(n)$证明命题$S(n+1)$，并称这种方法为“后靠”。有时候有人可能把这一过程看作从$S(n)$开始并证明$S(n+1)$，称这种方法为“前推”。在整数的情况下，这基本上是相同的意思。不过，对树而言，我们不能先假设命题对具有n个节点的树成立，并在某个位置加上一个节点，然后就说证明了结果对所有具有$n+1$个节点的树都成立。

例如，声明$S(n)$：“所有具有n个节点的树都有一条长度为$n-1$的路径。”$n=1$的依据情况显然为真。在错误的“归纳”中，可能会作出如下论证：“假设有一棵具有n个节点的树T，它有一条长$n-1$的路径，假如说是到节点v的。给v加上子节点u。现在就有了一棵具有$n+1$个节点的树，而且它有一条长度为n的路径，这样就证明了归纳步骤。”

当然，上述论证是谬误的，因为它没有证明结果对所有具有$n+1$个节点的树都为真，而只是证明了对选出的一些树为真。正确的证明不能是从n个节点“前推”到$n+1$个节点，因为我们不会从这一过程得出所有可能的树。我们必须从任意具有$n+1$个节点的树开始“后靠”，小心地选出一个节点，并将其删除，从而得到一棵具有n个节点的树。

---

① 分支系数和度是相关的概念，但它们是不同的，分支系数是树中各节点度的最大值。

5.6　二叉树

本节展现了另一种树——二叉树，它和5.2节介绍的“普通”树是不同的。在二叉树中，节点最多有两个子节点，而且并不是自然地从左起为子节点计数，而是有两个“槽”，其中一个用来存放左子节点，另一个用来存放右子节点。这两个槽中可能有一个为空，也可能两个都为空。



图5-27　两棵各只有两个节点的二叉树

✦ 示例 5.19

图5-27展示了两棵二叉树。它们均以n_1为根节点。第一棵树以n_2为左子节点，且没有右子节点。而第二棵树则没有左子节点，n_2是其根节点的右子节点。在这两棵树中，n_2都是既没有左子节点也没有右子节点。它们均为只有两个节点的二叉树。

接下来将按照如下方式递归地定义二叉树。

依据。空树是二叉树。

归纳。如果r是节点，而且T_1和T_2都是二叉树，那么以r为根节点，T_1为左子树，T_2为右子树可以组成一棵二叉树，如图5-28所示。也就是说，T_1的根节点就是r的左子节点，除非T_1是空树，因为这种情况下r没有左子节点。同样地，T_2的根节点是r的右子节点，除非T_2是空树，因为这种情况下r没有右子节点。

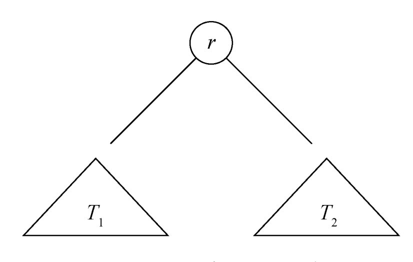


图5-28　二叉树的递归定义

5.6.1　二叉树的术语

5.2节引入的路径、祖先和子孙的概念也适用于二叉树。也就是说，左子节点和右子节点都归为“子节点”。路径仍然是节点m_1、m_2、…、m_k按照m_{i+1}是m_i（$i=1, 2, \cdots, k-1$）的（左或右）子节点的方式串联起来的序列。这条路径称为从m_1到m_k的路径。$k=1$的情况也是可以的，这样的话路径上就只有一个节点。

如果某个节点存在两个子节点，那么这两个子节点就互为兄弟节点。叶子节点是指既没有左子节点也没有右子节点的节点，还可以认为叶子节点是左子树和右子树都为空树的节点。内部节点则指不是叶子节点的节点。

路径长度、高度和深度的定义与普通的树是完全相同的。二叉树路径的长度要比节点数小1，也就是说，长度就是路径上父子关系对的数量。节点n的高度是指n到其子孙叶子节点最长路径的长度。二叉树的高度就是其根节点的高度。节点n的深度是指从根节点到n的路径的长度。

✦ 示例 5.20

图5-29展示了具有3个节点的二叉树可能形成的5种形状。在图5-29所示的每棵二叉树中，n_3都是n_1的子孙，并存在从n_1到n_3的路径。n_3在每棵树中都是叶子节点，而n_2在中间那棵树中是叶子节点，在其他4棵树中都是内部节点。

n_3在每棵树中的高度都为0，n_1除了在中间那棵树的高度为1之外，在其他树中的高度都为2。每棵树的高度都与n_1在那棵树中的高度一致。节点n_3除了在中间那棵树的深度为1之外，在其他树中深度都为2。

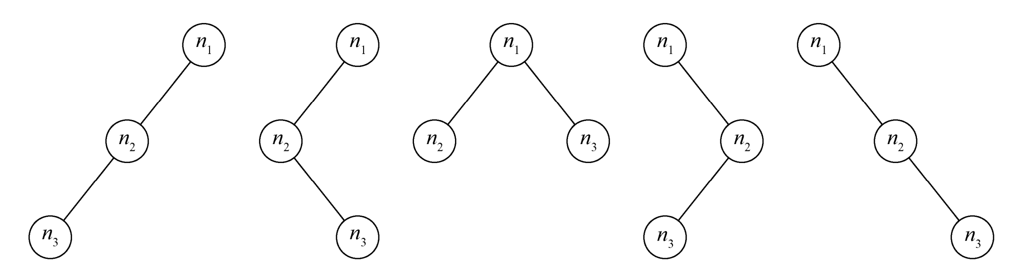
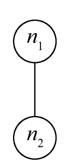


图5-29　具有3个节点的5种二叉树

（普通）树和二叉树的区别

尽管二叉树要求区分子节点是左子节点还是右子节点，但普通的树却没有这样的限制，理解这一点是很重要的。也就是说，二叉树不仅是节点的子节点数全都不多于两个的树。图5-27中的两棵树不仅是互不相同，而且与由根节点及根节点的子节点组成的如下普通树也没有关系。



这里还存在一个技术上的区别。尽管树的定义中表示树至少有一个节点，但二叉树中可以存在空树，也就是没有节点的树。

5.6.2　二叉树的数据结构

有一种很自然的方式可以用于表示二叉树。节点可以表示为具有`leftChild`和`rightChild`这两个分别指向左子节点和右子节点的字段的记录。出现在这两个字段中的`NULL`指针就表示对应的左子树或右子树为空——也就是说节点没有左子节点或右子节点。

二叉树可以表示为指向其根节点的指针。空二叉树很自然地就被表示为`NULL`。因此，如下类型定义就表示二叉树。

```
typedef struct NODE *TREE;
struct NODE {
    TREE leftChild, rightChild;
};
```

在这里，“指向节点的指针”类型名为TREE，因为这一类型最常见的用途就是表示树和子树。我们既可以将leftChild和rightChild字段解释为指向子节点的指针，也可以将其解释为指向左右子树本身的指针。

此外，还可以为表示NODE的结构体添加标号字段，并（或）可以添加指向父节点的指针。请注意，父指针的类型是*NODE，或是TREE的等价类型。

5.6.3　对二叉树的递归

有很多针对二叉树的自然算法可以通过递归的方式来定义。这里递归的模式要比图5-13中普通树的递归模式更具局限性，因为操作只能发生在左子树被探索之前、两棵子树的探索之间，或是两棵子树都探索完之后。对二叉树进行递归的模式如图5-30所示。

```
{
    action A0;
    对左子树的递归调用;
    action A1;
    对右子树的递归调用;
    action A2;
}
```

图5-30　二叉树递归算法的模板

✦ 示例 5.21

具有二元运算符的表达式树可以用二叉树表示。这些二叉树是很特殊的，因为节点要么有两个子节点，要么就没有子节点（一般而言，二叉树可以有只有一个子节点的节点）。例如，图5-31重现了图5-14中的表达式树，这棵表达式树可以视作二叉树。

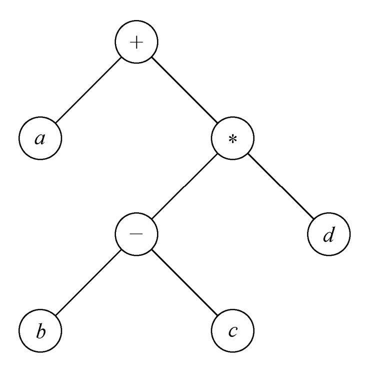


图5-31　由二叉树表示的表达式 $a+(b-c)*d$

假设为节点和树定义如下类型：

```
typedef struct NODE *TREE;
struct NODE {
    char nodeLabel;
    TREE leftChild, rightChild;
};
```

那么图5-32就展示了用来以前序列出二叉树T中各节点标号的递归函数。

```
     void preorder(TREE t)
     {
(1)      if (t != NULL) {
(2)          printf("%c\n", t->nodeLabel);
(3)          preorder(t->leftChild);
(4)          preorder(t->rightChild);
         }
     }
```

图5-32　二叉树的前序排列

该函数的行为与图5-15中用于处理普通树的同名函数相似。主要区别在于，当图5-32中的函数遇到叶子节点时，它会对（缺失的）左右子节点调用自身。这些调用会立即返回，因为当t为NULL时，整个函数体只有第(1)行的测试会执行。如果将图5-32中的第(3)行和第(4)行替换为：

```
(3)  if (t->leftChild != NULL) preorder(t->leftChild);
(4)  if (t->rightChild != NULL) preorder(t->rightChild);
```

就可以多节省一些调用。不过，这样就不能防止其他函数以NULL为参数调用preorder了。因此，为了安全起见，要保留第(1)行的测试。

5.6.4　习题

(1) 编写函数，使其能打印出二叉树节点（标号）的中序排列。假设这些节点是用如本节所描述那样具有左子节点和右子节点指针的记录表示的。

(2) 编写函数，使其接受二叉表达式树，并打印出它所表示的表达式带有全部括号的版本。假设这里使用了与习题(1)相同的数据结构。

(3) * 重复习题(2)，但只打印所需的括号，假设这里使用的是常用的算术运算符优先级和结合性。

(4) 编写函数，使其能得出二叉树的高度。

(5) 如果二叉树的节点同时具有左子节点和右子节点，那么就说该节点是*完全的*。用结构归纳法证明：二叉树中完全节点的数量，要比叶子节点的数量少1。

(6) 假设用左子节点右子节点记录类型表示二叉树。用结构归纳法证明：NULL指针的数量要比节点的数量大1。

(7) ** 树可以用来表示递归调用。每个节点都表示某个函数F的一次递归调用，而其子节点则表示F执行的调用。在本题中，要考虑对4.5节给出的$\binom{n}{m}$进行递归，根据的递归关系是$\binom{n}{m}=\binom{n-1}{m}+\binom{n-1}{m-1}$。每次调用都可以用一棵二叉树表示。如果某个节点对应着$\binom{n}{m}$的计算，而且不属于依据情况（$m=0$和$m=n$），那么其左子节点就表示$\binom{n-1}{m}$，而右子节点表示$\binom{n-1}{m-1}$。如果该节点表示的是依据情况，那么它就既没有左子节点，也没有右子节点。

(a) 通过结构归纳法证明：根节点对应着$\binom{n}{m}$的二叉树刚好有$2\binom{n}{m}-1$个节点。

(b) 利用(a)证明：$\binom{n}{m}$对应递归算法的运行事件是$O\left(\binom{n}{m}\right)$。请注意，该运行时间因此也就是$O(2^n)$，不过后者是平滑但非紧边界。

中序遍历

除了二叉树的前序排列和后序排列外，二叉树就只有一种有意义的节点排列方式了。在二叉树中，探索完左子树之后，而在探索右子树之前（即图5-30中操作T_1的位置）列出每个节点，这样就形成了二叉树节点的中序排列。例如，在图5-31所示的树中，中序排列就是$a+b-c*d$。

对表示表达式的二叉树进行前序遍历，得到的就是该表达式的前缀形式，对同样的树进行后序遍历，则会得到表达式的后缀形式。而中序遍历则几乎会产生原始的，或者说是中缀形式的表达式，不过该表达式是没有括号的。也就是说，图5-31中的树表示的表达式$a+(b-c)*d$与其中序排列$a+b-c*d$是不同的，差别只是后者中少了必要的括号而已。

要确保所需的括号出现，可以为所有的运算符加上括号。在这种修改后的中序遍历中，在探索左子树之前执行的操作A_0，会检查节点的标号是否为运算符，而且，如果是运算符的话，就会打印“(”，也就是左括号。同样地，如果标号是运算符，探索完两棵子树后执行的操作A_2就会打印右括号“)”。将该规则应用于图5-31所示的二叉树，得到的结果会是$\left(a+\left((b-c)*d\right)\right)$，这就有了$b-c$两侧一对必要的括号，以及两对多余的括号。

对二叉树进行结构归纳

与普通树一样，结构归纳法也适用于二叉树。其实还可以使用更简单的模式，这种模式下的依据是空树。下面就是对这一技巧的总结。

(1) 指定要证明的命题$S(T)$，其中T是一棵二叉树。

(2) 证明依据，即证明若T是空树，则$S(T)$为真。

(3) 设T是以r为根节点，并以T_L和T_R为子树的二叉树，以此构建归纳步骤。声明假定有归纳假设，即$S(T_L)$和$S(T_R)$为真。

(4) 在(3)中提到的假设下证明$S(T)$为真。

5.7　二叉查找树

各种计算机程序中有一种同样的活动，就是维护这样一组值，用户希望：

(1) 向这组值中插入元素；

(2) 从这组值中删除元素；

(3) 查找某元素，看看它是否在这组值中。

例子之一是英语词典，我们时不时地会往里面插入一些新单词，比如`fax`；删除一些不再使用的单词，比如`aegilops`；或者是要查找一串字母，看看其是否为单词，例如，这是拼写检查器程序的一部分。

因为这个例子是我们非常熟悉的，所以不管其具体用途是什么，只要是可以按照上述定义对其执行插入、删除和查找操作的一组值，都叫作词典。再举个词典的例子，某教授可能要记录选修某课程学生的花名册。偶尔会有学生被加入这门课程（插入），或是退出该课程（删除），或是需要弄清某个学生是否选修了该课程（查找）。

二叉查找树这种带标号的二叉树是实现词典的一个好方法。假设节点的标号是按照“小于”顺序（我们会将其写作 < ）从一组值中选出的。例子包括具有一般小于顺序的实数或整数，或是有着用 < 表示的词典顺序或字母表顺序的字符串。

二叉查找树（Binary Search Tree，BST）是一种带标号的的二叉树，以下属性对这种二叉树的每个节点x都成立：x的左子树中所有节点的标号都小于x的标号，而其右子树中所有节点的标号都大于x的标号。这种属性被称为**二叉查找树属性**。

✦ 示例 5.22

图5-33展示了对应着集合｛`Hairy`，`Bashful`，`Grumpy`，`Sleepy`，`Sleazy`，`Happy`｝的二叉查找树，其中 < 顺序是词典顺序。请注意，根节点左子树在词典顺序上都小于`Hairy`，而右子树在字典顺序上都大于它。这一属性对该树中的每个节点都成立。

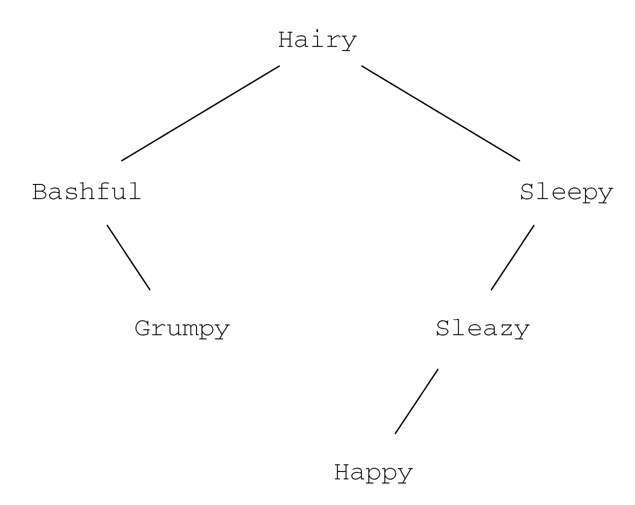


图5-33　具有6个带字符串标号的节点的二叉查找树

5.7.1　用二叉查找树实现词典

我们可以将二叉查找树表示为任何带标号的二叉树。例如，可以按如下形式定义`NODE`类型。

```
typedef struct NODE *TREE;
struct NODE {
    ETYPE element;
    TREE leftChild, rightChild;
};
```

二叉查找树被表示为指向二叉查找树根节点的指针。元素的类型`ETYPE`应该得到合理的设置。在本章所有的程序中，都将假设`ETYPE`为`int`类型，从而使元素间的比较可以直接用算术比较运算符`<`、`==`和`>`完成。在涉及词典顺序比较的例子中，可以假设程序中的比较由诸如2.2节中讨论过的*lt*、*eq*和*gt*这样的比较函数完成。

5.7.2　二叉查找树中元素的查找

假设想在由二叉查找树T表示的某词典中查找某个元素x。如果将x与T的根节点处的元素加以比较，就可以利用二叉查找树属性快速找到x，或确定x没有出现。如果x在根节点处，就完成了查找。否则，如果x比根节点处的元素小，x就只可能在左子树中被找到（根据二叉查找树属

性），而如果x比根节点处的元素大，x就只可能出现在右子树中（还是因为二叉查找树属性）。也就是说，可以通过以下递归算法表示查找操作。

依据。如果树T是空树，那么x未出现。如果树T非空，而且x在根节点，那么x就出现了。

归纳。如果T非空而x未在根节点位置，设y是T根节点处的元素。如果 $x<y$ ，则只在根节点的左子树中查找x；如果 $x>y$ ，则只在y的右子树中查找x。二叉查找树属性保证x不可能出现在没有查找的那棵子树中。

抽象数据类型

诸如插入、删除和查找这种可能对一组对象或特定类别执行的一系列操作，有时称为抽象数据类型或ADT。这一概念也会被称为类或模块。我们将在第7章中研究若干抽象数据类型，而在本章中，我们会看到其中一个：优先级队列。

ADT可以有多种抽象实现。例如，在本节中可看到二叉查找树是一种实现词典ADT的好方法。表是另一种看似可靠实则经常效率低下的实现词典ADT的方式。7.6节将介绍散列，另一种不错的词典实现方式。

每种抽象实现依次可通过若干种不同的数据类型来具体实现。举例来说，可以使用二叉树的左子节点右子节点实现作为实现二叉查找树的数据结构。这一数据结构，加上用于插入、删除和查找的恰当函数，就成了词典ADT的一种实现。

在程序中使用ADT的一个重要原因是，ADT底层的数据只能通过ADT的操作（比如插入）来访问。这一限制是防御性编程的一种形式，可以防止操作数据的函数以意料之外的方式对数据进行偶发变更。使用ADT的第二个重要原因在于，ADT让我们可以重新设计数据结构和实现其操作的函数，在不担心会为程序其余部分引入错误的前提下，这样可能提高操作的效率。如果只有用于ADT操作的接口函数被正确地重写，就不会出现新的错误。

✦ 示例 5.23

假设想要在图5-33的二叉查找树中查找`Grumpy`。将`Grumpy`与根节点处的`Hairy`相比较，发现`Grumpy`在词典顺序上要先于`Hairy`，因此要对左子树调用`lookup`。

左子树的根节点是`Bashful`，而将该标号与`Grumpy`相比，发现前者要先于后者。因此要对`Bashful`的右子树递归地调用`lookup`。现在发现`Grumpy`在这一子树的根节点处，并返回`TRUE`。这些步骤是由具有图5-34模式的词典顺序比较函数执行的。

```
     BOOLEAN lookup(ETYPE x, TREE T)
     {
(1)      if (T == NULL)
(2)          return FALSE;
(3)      else if (x == T->element)
(4)          return TRUE;
(5)      else if (x < T->element)
(6)          return lookup(x, T->leftChild);
         else /* x 一定是大于T->element */
(7)          return lookup(x, T->rightChild);
     }
```

图5-34 如果x在T中，函数`lookup(x,T)`会返回`TRUE`，否则返回`FALSE`

更具体地讲，图5-34中的递归函数lookup(x,T)使用左子节点右子节点数据结构实现了这一算法。请注意，lookup返回的是BOOLEAN类型的值，这一类型实际上与int相同，不过它定义的值只有TRUE和FALSE，分别定义为1和0。BOOLEAN类型是在1.6节中引入的。此外，请注意，这里的lookup函数只接受能由=、<等运算符比较的类型。如果要让它处理示例5.23中用到的字符串那样的数据，就需要重写。

在第(1)行，lookup会确定T是否为空。如果不为空，那么lookup在第(3)行会确定x是否存储在当前节点。如果x不在该节点，那么lookup就会根据x是小于还是大于当前节点存储的元素，递归地查找左子树或右子树。

5.7.3　二叉查找树元素的插入

向二叉查找树T中增加一个新元素x是很简单的，以下递归算法简要描述了处理思路。

依据。如果T是空树，用一棵由单个节点构成的树替代T，并在该节点处放上x。如果T非空而且其根节点处有元素x，那么x已经在字典中，不需要再做任何事情。

归纳。如果T非空，而且x不在其根节点处，那么如果x小于根节点处的元素，就将x插入左子树，如果x大于根节点处的元素，就将x插入右子树。

如图5-35所示的insert(x,T)函数为左子节点右子节点数据结构实现了这一算法。在第(1)行发现T的值为NULL时，就会新建一个节点，该节点就成了树T。第(2)到第(5)行会创建该树，并在第(10)行返回。

```
        TREE insert(ETYPE x, TREE T)
        {
(1)         if (T == NULL) {
(2)             T = (TREE) malloc(sizeof(struct NODE));
(3)             T->element = x;
(4)             T->leftChild = NULL;
(5)             T->rightChild = NULL;
            }
(6)         else if (x < T->element)
(7)             T->leftChild = insert(x, T->leftChild);
(8)         else if (x > T->element)
(9)             T->rightChild = insert(x, T->rightChild);
(10)        return T;
        }
```

图5-35　insert(x,T)函数将x添加到T中

如果在T的根节点处没找到x，那么在第(6)到第(9)行，会根据相应的情况对左子树或右子树调用insert函数。被该插入操作修改过的子树，会分别在第(7)或第(9)行成为T的根节点的左子树或右子树的新值。第(10)行会返回增加过元素的树。

请注意，如果x在T的根节点，那么第(1)、第(6)和第(8)行的测试都不会成功。这种情况下，insert会在什么都不做的情况下返回T，这是正确的，因为x已经在树中了。

✦ 示例 5.24

继续示例5.23的问题，从技术上理解，对字符串进行比较需要的代码与图5-35相比稍有不同，图5-35中<这样的算术比较要替代为对*lt*这样恰当定义的函数的调用。图5-36展示了向图5-33插入Filthy后的二叉查找树。首先对根节点调用insert，发现Filthy<Hairy。因此，在图

5-35的第(7)行，对左子节点调用insert。结果发现Filthy>Bashful，所以接着在第(9)行对右子节点调用insert。这样就到了Grumpy，它从词典顺序上在Filthy之后，因此就要对Grumpy的左子节点调用insert。

指向Grumpy左子节点的指针为NULL，所以在第(1)行必须创建一个新节点。这棵单节点树会返回给Grumpy节点处对insert的调用，而且在第(7)行该树会被安置为Grumpy左子节点的值。带有Grumpy和Filthy的修改过的树会返回给标号为Bashful的节点处对insert的调用。然后，以Bashful为根节点的新树就成了整棵树根节点的左子树。最终的树如图5-36所示。

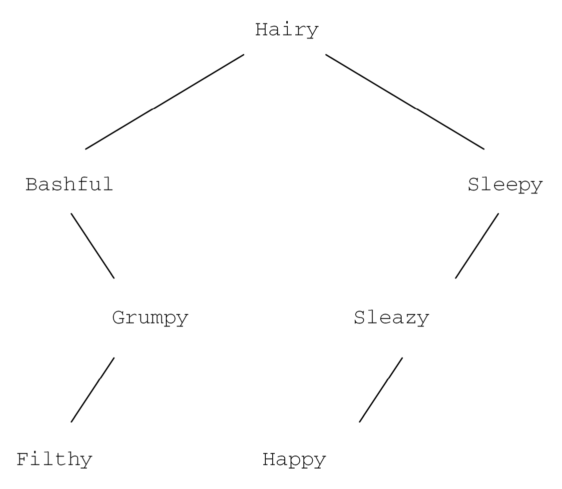


图5-36　插入Filthy之后的二叉查找树

5.7.4　二叉查找树元素的删除

从二叉查找树中删除某个元素x要比查找或插入复杂一些。首先，要找出含有x的节点；如果没有这样的节点，就算是完事了，因为x不在这棵要处理的树里头。如果x在叶子节点处，那么直接删除该叶子节点就行了。不过，如果x是某个内部节点n，就不能直接删除该节点，因为这样做会破坏树的连通性。

我们必须以某种方式对树进行重新排列，从而在维持二叉查找树属性的同时让x从树中消失。这会有两种情况。第一种，如果n只有一个子节点，就用该子节点代替n，这样二叉查找树的属性就得到了保持。

第二种情况，假设n的两个子节点都存在。一种策略就是找到标号为y的节点m，它是n右子树中最小的元素，并在节点n处用y代替x，如图5-37所示。然后就可以从右子树中删除节点m。

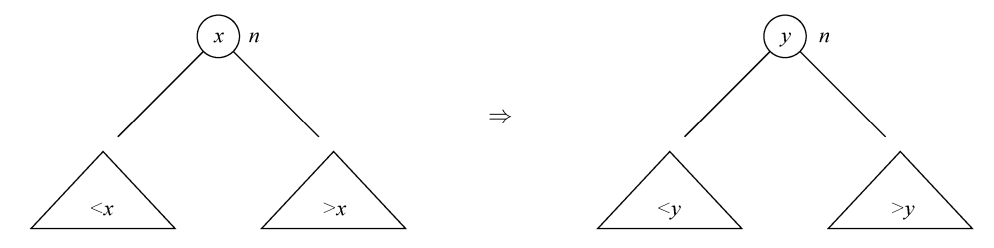


图5-37　要删除x，先删除包含右子树中最小元素y的节点，然后将节点n处的标号由x替换为y

此时二叉查找树属性继续成立。原因在于，x比n的左子树中的所有元素都大，而y大于x（因为y在n的右子树中），所以y也比n左子树的所有元素都大。因此，就n的左子树而言，y是适合于位置n的元素。而对n的右子树来说，y也是适合作为根节点的，因为选出的y是右子树中的最小元素。

```
     ETYPE deletemin(TREE *pT)
     {
         ETYPE min;

(1)      if ((*pT)->leftChild == NULL) {
(2)          min = (*pT)->element;
(3)          (*pT) = (*pT)->rightChild;
(4)          return min;
         }
         else
(5)          return deletemin(&((*pT)->leftChild));
     }
```

图5-38　`deletemin`（`pT`）函数会删除并返回T的最小元素

如图5-38所示，可以很方便地定义`deletemin(pT)`函数从非空二叉查找树中删除含最小元素的节点，并返回最小元素的值。我们给该函数传入的参数是指向树T的指针的地址。该函数中所有对T的引用都是通过该指针间接完成的。

我们给函数传入的参数，是指向某个位置的指针，而在这个位置可以找到指向节点（即树）的指针，这种风格的树操作叫作*按引用调用*。这在图5-38中是很关键的，因为第(3)行的指针指向左子节点为`NULL`的节点m，我们希望将这一指针替代为另一个指针，节点m的`rightChild`字段中的指针。如果`deletemin`的参数是指向节点的指针，那么这种改变就会在对`deletemin`的调用中发生，而且树中的指针其实不会改变。顺便说一下，也可以使用按引用调用来实现插入操作。在那种情况下，可以直接对树进行修改，而不必像图5-35中所做的那样返回修改过的树。这里将这一修订过的`insert`函数留作本节习题。

现在来看看图5-38的工作原理。沿着左子节点向下寻找，直到在图5-38的第(1)行找到左子节点为NULL的节点，就找到了最小的元素。在该节点m处的元素y一定是该子树中最小的元素。原因在于，这里完全是循着左子节点向下寻找的，这样一来y要比m在该子树中的任一祖先都小。而子树中其他节点要么是在m的右子树中，根据二叉查找树属性这些元素肯定大于y，要么是在m的某个祖先的右子树中。右子树中的元素肯定比m的某个祖先处的元素大，因此也就大于y，如图5-39所示。

在子树中找到最小的元素后，在第(2)行会记录下它的值，并在第(3)行用它的右子树代替最小元素所在节点。请注意，在从子树中删除最小元素时，总是有着最简单的删除情况，因为不存在左子树。

还有一点与`deletemin`相关的内容就是，当第(1)行的测试失败时，就意味着还没到达最小元素，就要继续处理左子节点。这一步骤是通过第(5)行的递归调用完成的。

`deletemin(x,pT)`函数如图5-40所示。如果`pT`指向空树T，就没什么要做的，而且第(1)行的测试会确保什么事都没做。此外，第(2)行和第(4)行的测试会处理x不在根节点的情况，会根据具体情况重定向到左子树或右子树。如果到达第(6)行，那么x就一定在T的根节点位置，而且我们必须替换该根节点。第(6)行会测试左子节点是否可能为`NULL`，若为`NULL`，那么就可以在第(7)行直接将T替换为其右子树。同样地，如果在第(8)行发现右子节点为`NULL`，那么就用T的左子

树来替代T。请注意，如果根节点的两个子节点都为NULL，那么就在第(7)行将T替换为NULL。

两个子节点均不为NULL的情况是在第(10)行处理的。在这里会调用deletemin，返回右子树的最小元素y，并从该子树中删除y。第(10)行的赋值操作会在T的根节点处用y代替x。

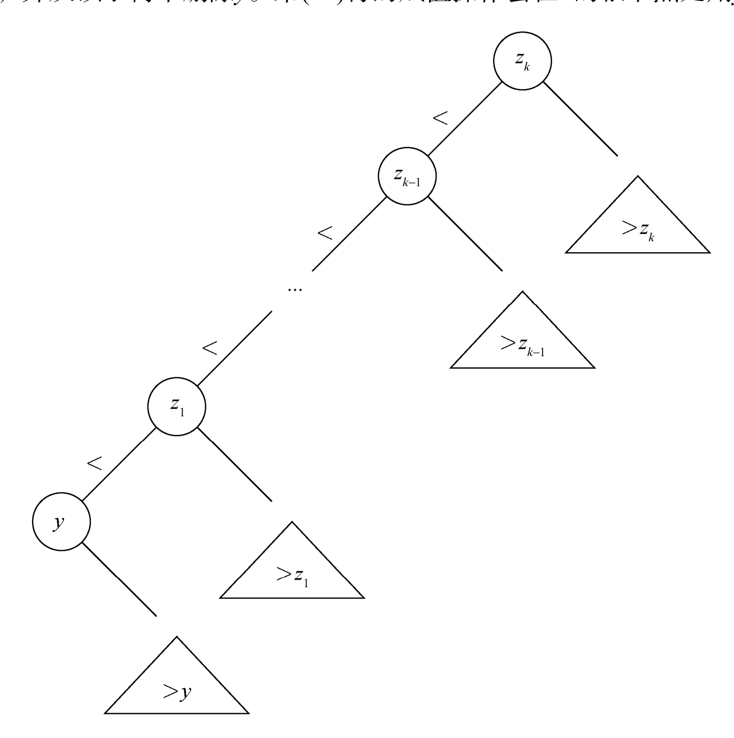


图5-39　右子树中所有其他元素都大于y

```
      void delete(ETYPE x, TREE *pT)
      {
(1)       if ((*pT) != NULL)
(2)           if (x < (*pT)->element)
(3)               delete(x, &((*pT)->leftChild));
(4)           else if (x > (*pT)->element)
(5)               delete(x, &((*pT)->rightChild));
              else /* 这里，x在(*pT)的根节点处 */
(6)               if ((*pT)->leftChild == NULL)
(7)                   (*pT) = (*pT)->rightChild;
(8)               else if ((*pT)->rightChild == NULL)
(9)                   (*pT) = (*pT)->leftChild;
                  else /* 这里的两个子节点都不为NULL */
(10)                  (*pT)->element =
                              deletemin(&((*pT)->rightChild));
      }
```

图5-40　delete(x,pT)函数从T中删除元素x

✦ 示例 5.25

如果使用类似delete（但能够比较字符串）的函数从图5-36中的二叉查找树删除Hairy，结果如图5-41所示。因为Hairy在具有两个子节点的节点中，所以delete会调用deletemin

函数，从根节点的右子树中删除并返回最小的元素Happy，然后Happy就成了曾存放Hairy的该树根节点的标号。

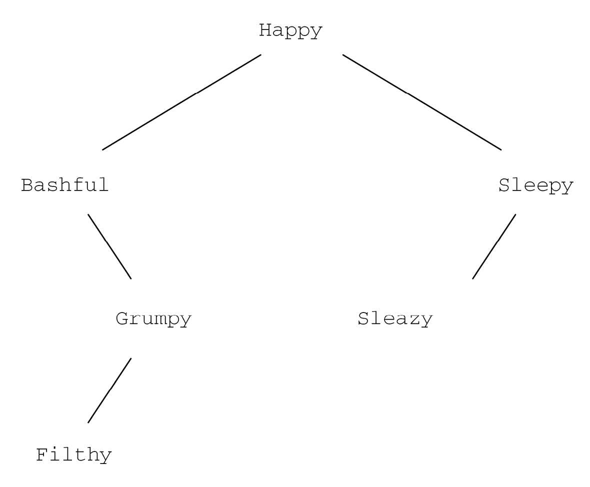


图5-41　删除Hairy后的二叉查找树

5.7.5　习题

(1) 假设使用最左子节点右兄弟节点表示法实现二叉查找树。重新编写适用于这一数据结构的实现了插入、删除和查找这些词典操作的函数。

(2) 如果按顺序插入单词：Doc、Dopey、Inky、Blinky、Pinky和Sue，会使图5-33中的二叉查找树变成什么样？然后，依次删除Doc、Sleazy和Hairy后又会怎样？

(3) 用对字符串的词典比较代替对整数的算术比较，重新编写lookup、insert和delete函数。

(4) * 重新编写insert函数，使得树参数可以按引用传递。

(5) * 在本节中，我们曾以“按引用调用”的方式编写过delete函数。不过，也可以用编写insert函数的风格编写该函数，即接受树作为参数，而不是接受指向树的指针作为参数。编写这一版本的delete操作。注意：让deletemin返回修改过的树并非真正可能，因为它还必须返回最小的元素。我们可以重新编写deletemin，使其返回同时具有新树和最小元素的结构体。

(6) 要删除带有两个子节点的节点，除了通过在右子树中找到最小元素，还可以在左子树中找到最大的元素，并用它替代删除的元素。重新编写来自图5-38和图5-40的delete和deletemin函数，从而融入这种修改。

(7) * 在需要删除某个具有父节点p、（非空的）左子节点l和（非空的）右子节点r的节点n处的元素时，另一种处理删除操作的方式是，找出n的右子树中存放最小元素的节点m。接着，让r代替n成为p的左子节点或右子节点，并让l成为m的左子节点。请注意，m之前不能有左子节点。证明这一系列改变为何会保留二叉查找树属性。大家是否愿意选择这一策略替代5.7节中描述过的那种？提示：对这两种方式而言，考虑它们对路径长度的影响。正如我们将在5.8节中看到的，短路径会让操作运行得更迅速。

(8) * 在本习题中，考虑图5-39所示的二叉查找树。通过对i的归纳证明，如果$1 \leqslant i \leqslant k$，那么$y < z_i$。然后，证明$y$是以$z_k$为根节点的树中最小的元素。

(9) 编写完整的C语言程序，实现存储整数的词典。接受形为x i的命令，其中x是字母i（插入）、d（删除）和l（查找）中的一个。整数i是该命令的参数，就是有待插入、删除或查找的整数。

5.8　二叉查找树操作的效率

二叉查找树提供了一种相当快速的词典实现。首先请注意，插入、删除和查找操作各会进行若干次递归调用，调用次数等于所经过的路径的长度。但该路径必须包含达到右子树最小元素的路线，以防`deletemin`被调用。对`lookup`、`insert`、`delete`和`deletemin`函数进行简单的分析，可知各操作都花费$O(1)$的时间，而且要加上一次递归调用的时间。此外，因为该递归调用总是在当前节点的子节点处进行的，所以每次成功调用中节点的高度至少要减少1。

因此，如果以指向某个高度为h的节点的指针调用这些函数所花的时间为$T(h)$，就有以下递推关系来为$T(h)$确定上界。

依据。$T(0)=O(1)$。也就是说，在对叶子节点调用函数时，该调用要么终止，不再有进一步的调用，要么以`NULL`参数进行一次递归调用，接着会返回而不再继续调用。这些工作所花时间为$O(1)$。

归纳。对$h\geqslant 1$，$T(h)\leqslant T(h-1)+O(1)$。也就是说，对任何内部节点调用函数所花的时间，都等于$O(1)$加上对高度至多为$h-1$的节点进行一次递归调用所花的时间。如果作出$T(h)$会随着h的增加而增加这一合理假设，那么该递归调用的时间不会大于$T(h-1)$。

该递推关系的解是$O(h)$，正如3.9节中讨论过的那样。因此，对具有n个节点的二叉查找树执行词典操作的运行时间至多与该树的高度成比例。不过具有n个节点的二叉查找树通常高度为多少呢？

5.8.1　最坏情况

在最坏的情况下，二叉树的所有节点都排列在一条路径上，就像图5-42所示的树那样。例如，取一列有序的k个元素，将这些元素一个个地依次插入一棵空树，就可以形成这样的树。也有不全由右子节点组成，而是由左右子节点混合组成的单路径树，其路径上的内部节点既可能是左子节点也可能是右子节点。

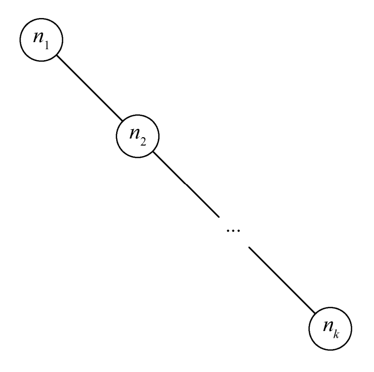


图5-42　退化的二叉树

像图5-42这样具有k个节点的树的高度显然为$k-1$。因此可以预见，如果具有k个元素的词典的表示不幸是这些树中的一种，那么查找、插入和删除操作所花时间都会是$O(k)$。从直觉上讲，如果需要查找某个元素x，平均需要走过一半路径才会找到它，需要查看$k/2$个节点。如果这还没有找到x，就需要继续向下搜索该树，直到到达x所在的位置为止，平均也要走过该路线

的一半。因为查找、插入和删除操作都涉及元素的查找，所以可知，在给定图5-42所示树的最坏情况下，这些操作平均要花的时间都是$O(k)$。

5.8.2　最佳情况

不过，二叉树不一定非得像图5-42这样又高又瘦，它也可以是图5-43这种分枝丛生的低矮样式。而后者这样的树，每个内部节点向下到某层的两个子节点都存在，而且下一层的所有叶子节点也都存在，这样的结构叫作完全树或完整树。

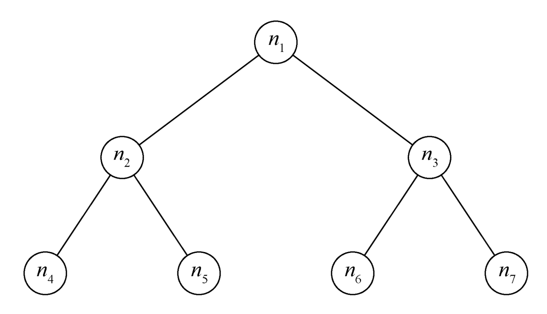


图5-43　有7个节点的完全二叉树

高度为h的完全二叉树共有$2^{h+1}-1$个节点。我们可以通过对高度h的归纳证明这一声明。

依据。如果$h=0$，那么该树由一个节点组成。因为$2^{0+1}-1=1$，所以依据情况成立。

归纳。假设高度为h的完全二叉树有$n^{h+1}-1$个节点，并考虑高度为$h+1$的完全二叉树。该树有一个根节点，并由两棵高度为h的完全二叉树分别作为其左右子树。例如，图5-43中高度为2的完全二叉树包含根节点n_1，由n_2、n_4和$n_5$3个节点构成的左子树（高度为1的完全二叉树），以及由其余3个节点构成的右子树（另一棵高度为1的完全二叉树）。根据归纳假设，两棵高度为h的完全二叉树共有$2(2^{h+1}-1)$个节点。在加上根节点后，可知高度为$h+1$的完全二叉树共有$2(2^{h+1}-1)+1=2^{h+2}-1$个节点，这就证明了归纳步骤。

现在可以将这一关系反转，说一棵具有$k=2^{h+1}-1$个节点的完全二叉树高度为h。这样一来，$k+1=2^{h+1}$。两边取对数，就有$\log_2(k+1)=h+1$，或者大致可以说h是$O(\log k)$。因为查找、插入和删除的运行时间都与树的高度成比例，所以这些操作所花的时间是节点数的对数。这样的性能要比图5-42所示最糟情况所花的线性时间强多了。随着词典的大小越来越大，词典操作运行时间的增长要比集合中元素的增长慢得多。

5.8.3　一般情况

图5-42所示情况和图5-43所示情况哪个更普遍？其实，两者在实践中都不常见，不过图5-43中的完全树提供的词典操作效率与一般情况的效率是近似的。也就是说，平均情况下，查找、插入和删除花费的都是对数时间。

要证明一般二叉树可以提供对数时间的词典操作是很难的。证明的要点在于，从这样的树的根节点到某个随机节点的路径长度的期望值是$O(\log n)$。本节习题中将给出这一期望值的递推等式。

不过，我们可以直观地看出这为什么应该是正确的运行时间，理由如下。二叉树的根节点会将除它自己之外的节点分为两棵子树。在最平均的分布中，一棵有k个节点的树将会有两棵各

有约 $k/2$ 个节点的子树。如果根节点元素刚好是在一列有序元素的正中位置，就会形成这种情况。而在最坏的分布中，根节点元素是词典中的第一个或最后一个元素，这样就会使一棵子树为空，而另一棵子树中有 $k-1$ 个元素。

平均而言，可以预期根节点是在已排序表正中和极端之间，而且可以预期约有 $k/4$ 个节点在较小的子树中，另外的 $3k/4$ 个节点在较大子树中。假设在向下探索树的过程中，每次递归调用时始终移动到较大子树的根节点，而且类似的假设适用于每层元素的分布。在第一层，较大子树会按照1∶3的比例划分，在第二层留下共有 $(3/4)(3k/4)$，即 $9k/16$ 个节点的最大子树。因此，可以预见在第d层的最大子树约有 $(3/4)^d k$ 个节点。

如果d变得足够大，那么 $(3/4)^d k$ 的量会接近1。而且可以预见，在这一层，最大子树将是由一个叶子节点组成的。因此要问，d为什么值可以使 $(3/4)^d k \leqslant 1$？如果取以2为底的对数，就得到

$$d\log_2(3/4)+\log_2 k \leqslant \log_2 1 \tag{5.1}$$

现有 $\log_2 1=0$，且 $\log_2(3/4)$ 是一个负常数，约为 -0.4。因此可将(5.1)式重新写为 $\log 2k \leqslant 0.4d$，或 $a \geqslant (\log_2 k)/0.4 = 2.5\log_2 k$。

换句话说，在深度约为节点数以2为底的对数的2.5倍的位置（或是在更高的层数），就有望全是叶子节点了。这一论述证实了（但并未证明）一般二叉查找树的高度与该树节点数的对数成正比这一陈述。

5.8.4 习题

(1) 如果树T高度为h，而且分支系数为b，那么树T最多可以有多少个节点，最少有多少个节点？

(2) ** 进行如下实验，从n个不同值的$n!$种顺序中任选一种，并按照这一顺序将这些值插入一棵空的二叉查找树中。设 $P(n)$ 是实验后这n个值中某个特定值v所在节点的深度的期望值。

(a) 证明，对 $n \geqslant 2$，

$$P(n)=1+\frac{2}{n^2}\sum_{k=1}^{n-1}kP(k)$$

(b) 证明 $P(n)$ 是 $O(\log n)$。

5.9 优先级队列和偏序树

到目前为止，我们只看到一种抽象数据类型——词典，以及它的一种实现——二叉查找树。本节将研究另一种抽象数据类型以及它最有效率的一种实现。这种叫作*优先级队列*的抽象数据类型是各自有优先级与之关联的一组元素。例如，这些元素可以是一些记录，而优先级则可能是记录中某个字段的值。与优先级队列ADT有关的两种操作如下：

(1) 向集合中插入一个元素（`insert`）；

(2) 从集合中找出优先级最高的元素并将其删除（这种组合操作称为`deletemax`），被删除的元素由该函数返回。

✦ 示例 5.26

分时操作系统从多个来源接受服务请求，而这些作业的优先级可能不尽相同。例如，优先级最高的可能是系统进程，这些进程中可能包含监控传入数据（比如在终端的按键动作生成的

信号，或是局域网上数据包的到达所生成的信号）的“守护进程”。接着可能是用户进程，那些由普通用户发出的指令。再下来就可能是某些特定的后台作业，比如向磁带备份数据，或是用户已指定以低优先级运行的长计算。

作业可以表示为记录，这种记录由对应作业的整数ID和对应作业优先级的整数组成。也就是说，可以使用如下结构体

```
struct ETYPE {
    int jobID;
    int priority;
};
```

表示优先级队列中的元素。在初始化新的作业时，它会得到一个ID和一个优先级。然后对等待服务的作业构成的优先级队列执行这一元素的*插入*操作。当处理器资源可用时，系统就会来到优先级队列，并执行`deletemax`操作。由该操作返回的元素就是等待服务的作业中优先级最高的作业，而该作业正是接下来要执行的。

✦ 示例 5.27

我们可以使用优先级队列ADT实现排序算法。假设有一列整数a_1、a_2、…、a_n要排序，可以将这些整数放入一个优先级队列，分别使用这些元素的值作为各自的优先级。如果随后执行`deletemax`操作n次，这些整数就会按照从大到小的顺序依次被选出来。5.10节还会更详细地讨论这种称为堆排序的算法。

5.9.1 偏序树

实现优先级队列的一种有效方式是使用*偏序树*（Partially Ordered Tree，POT），这是一种具有如下属性的带标号二叉树。

(1) 节点的标号是具有“优先级”的元素，该优先级可以是元素的值，也可以是元素某个组成部分的值。

(2) 存储在节点中的元素的优先级，不小于存储在其子节点中的元素的优先级。

属性(2)说明，任何子树根节点处的元素总是该子树中最大的元素。我们将属性(2)称为*偏序树属性*，或*POT属性*。

✦ 示例 5.28

图5-44展示了一棵具有10个元素的偏序树。在这里以及本节的其他部分中，我们都将用元素的优先级来表示它们，就像元素和它们的优先级是一回事那样。请注意，相等的元素可能出现在树中的不同层级。要说明偏序树属性在根节点得到满足，请注意，根节点处的元素18不小于其子节点处的元素18和16。同样，可以验证在该树的每个内部节点处偏序树属性都成立。因此，图5-44是一棵偏序树。

偏序树为优先级队列提供了一种实用的抽象实现。简单地说，要执行`deletemax`操作，就要找到根节点，它肯定是最大的，并用底层的最右节点代替它。不过，这样做时，偏序树属性可能被破坏，因此必须还原偏序树属性，要让新放置在根节点处的元素“向下沉”，直到它到达合适的层次，使得它小于它的父节点，并且不小于它的子节点。要执行*插入*操作，就要在底层尽可能左的位置增加一个新的叶子节点，如果底层没有空位置，就要新添加一个层级，并将该节点放在新一层的左端。这样也可能对偏序树属性造成破坏，如果造成破坏，就要让新元素“向上冒”，直到它找到合适的位置。

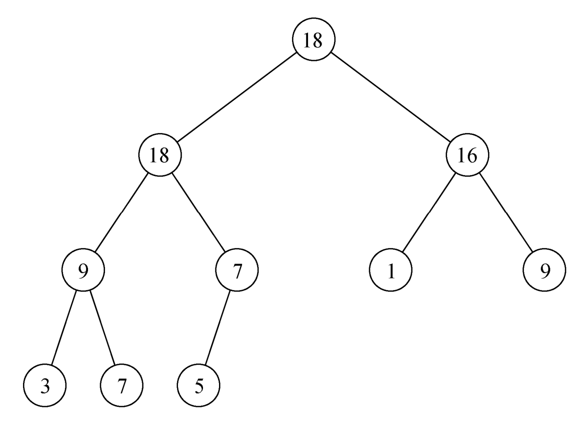


图5-44　有10个节点的偏序树

5.9.2　平衡偏序树和堆

如果偏序树除最下层之外的所有层级的节点全存在，而且最下层的叶子节点尽可能集中在左侧，那么这样的偏序树就是平衡的。这一条件说明，如果该树有n个节点，那么从根节点到任何节点的路径都不可能比$\log_2 n$长。图5-44中的树就是平衡偏序树。

平衡偏序树可以用称为堆的数组数据结构实现，这种数据结构提供了一种迅速、紧凑的优先级队列ADT实现。堆就是对元素下标有着特殊解释的数组A。首先从$A[1]$中的根节点开始，并未使用$A[0]$。在根节点之后，各层级依次出现。在同一层级中，节点按照从左到右的顺序排列。

因此，根节点的左子节点是在$A[2]$中，而根节点的右子节点在$A[3]$中。一般而言，$A[i]$处节点的左子节点在$A[2i]$中，而其右子节点在$A[2i+1]$中，如果这些子节点在偏序树中都存在的话。这种树的平衡性质使这种表示成为可能。这些元素的偏序树属性说明，如果$A[i]$有两个子节点，那么$A[i]$不小于$A[2i]$和$A[2i+1]$，如果$A[i]$只有一个子节点，那么$A[i]$不小于$A[2i]$。

1	2	3	4	5	6	7	8	9	10
18	18	16	9	7	1	9	3	7	5

图5-45　图5-44对应的堆

实现的层次

对词典和优先级队列这两种ADT进行比较，并注意到每种情况下只给出了一种抽象实现以及对应该抽象实现的一种数据结构，这样做是很有意义的。每种ADT都有其他的抽象实现，而每种抽象实现也都有其他的数据结构。之前已经说过，在本书随后的内容中还将讨论词典的其他抽象实现，比如散列表，而且在5.9节的习题中表示过，二叉查找树对优先级队列而言也是种合适的抽象实现。下表总结了目前为止我们对词典和优先级队列的抽象实现及数据结构的了解。

ADT	抽象实现	数据结构
词典	二叉查找树	左子节点右子节点结构
优先级队列	平衡偏序树	堆

✦ 示例 5.29

表示图5-44所示平衡偏序树的堆如图5-45所示。例如，$A[4]$存放着值9，这一数组元素表示图5-44中根节点的左子节点的左子节点。而该节点的子节点则在$A[8]$和$A[9]$中。它们的元素分别是3和7，都不大于9，正如偏序树属性所要求的。数组元素$A[5]$对应着根节点左子节点的右子节点，它的左子节点在$A[10]$的位置。它可以有存放在$A[11]$中的右子节点，但图5-44中的偏序树只有10个元素，所以$A[11]$并不是该堆的一部分。

尽管这里展示的树节点和数组元素似乎就只是优先级本身，但原则上讲树节点或数组中出现的是完整的记录。正如我们将要看到的，在偏序树或其堆表示的父子节点间要进行很多元素交换。因此，如果数组元素是指向表示优先级队列中各对象的记录的指针，并将这些记录存储在堆“之外”的另一个数组中，就会更有效率。这样就可以在不调整记录本身的情况下直接对指针进行交换。

5.9.3　优先级队列操作在堆上的执行

在5.9节和5.10节中，会用全局整数数组`A[1..MAX]`表示堆。这里假设元素都是整数，而且都等于它们的优先级。当元素是记录时，可将指向记录的指针存储在数组中，并根据记录中的某个字段来确定元素的优先级。

假设有一个满足偏序树属性的具有$n-1$个元素的堆，我们要向$A[n]$中添加第n个元素。偏序树属性在各处都继续成立，除了在$A[n]$和它的父节点间可能有例外。因此，如果$A[n]$大于其父节点位置的元素$A[n/2]$，就必须交换这些元素。而$A[n/2]$与其父节点间也可能违背偏序树属性。如果这样的话，就要让新元素递归地“冒泡”，直到它到达父节点有一个更大元素的位置，或是到达根节点位置。

执行这一操作的C语言函数`bubbleUp`如图5-46所示。它使用`swap(A,i,j)`函数交换$A[i]$和$A[j]$处的元素，该函数也是在图5-46中定义的。`bubbleUp`的操作很简单。给定表示节点的参数i，它表示的节点与其父节点有可能违背偏序树属性，测试是否有$i=1$，也就是测试是否为根节点，在根节点的话就不会破坏偏序树属性。如果不是，则测试$A[i]$是否大于其父节点处的元素；如果是，就在其父节点处递归地调用`bubbleUp`，交换$A[i]$与其父节点。

```
void swap(int A[], int i, int j)
{
    int temp;

    temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

void bubbleUp(int A[], int i)
{
    if (i > 1 && A[i] > A[i/2]) {
        swap(A, i, i/2);
        bubbleUp(A, i/2);
    }
}
```

图5-46　`swap`函数交换数组元素，而`bubbleUp`函数则将堆中的新元素推到它的右侧位置

✦ 示例 5.30

假设有图5-45所示的堆，并向其添加了优先级为13的第11个元素。该元素会出现在$A[11]$中，就有了如下数组

1	2	3	4	5	6	7	8	9	10	11
18	18	16	9	7	1	9	3	7	5	13

调用`bubbleUp(A,11)`，对$A[11]$和$A[5]$进行比较，因为$A[11]$更大，所以必须交换这两个元素。也就是说$A[5]$和$A[11]$违背了偏序树属性。因此，数组就成了

1	2	3	4	5	6	7	8	9	10	11
18	18	16	9	13	1	9	3	7	5	7

调用`bubbleUp(A,5)`，对$A[2]$和$A[5]$进行比较，因为$A[2]$更大，所以不会违背偏序树属性，`bubbleUp(A,5)`不会进行任何操作。这样就已经恢复了该数组的偏序树属性。

现在介绍如何实现优先级队列的插入操作。设n是优先级队列中当前的元素数，并假设数组`A[1..n]`已经满足偏序树属性。增加n，并将待插入的元素存储到新的`A[n]`中。最后，调用`bubbleUp(A,n)`。表示插入操作的代码如图5-47所示。参数x是待插入的元素，而参数`pn`是指向优先级队列当前大小的指针。请注意，n必须按引用传递，也就是说，通过指向n的指针传递，这样当n增加时，改变才不只是在插入操作局部造成影响。这里省略了对 $n<MAX$ 的检查。

```
void insert(int A[], int x, int *pn)
{
    (*pn)++;
    A[*pn] = x;
    bubbleUp(A, *pn);
}
```

图5-47　在堆上实现的优先级队列插入操作

要实现优先级队列的`deletemax`操作，需要对堆或偏序树进行另一项操作，这次是让根节点处可能违背偏序树属性的元素向下沉。假设$A[i]$可能违背偏序树属性，在它中的元素可能小于其子节点$A[2i]$和$A[2i+1]$中的一个或两个。我们可以将其与其中一个子节点交换，不过一定要注意是与哪一个交换。如果与两个子节点中较大的那个交换，那么肯定不会在$A[i]$曾经的两个子节点间引入偏序树属性的破坏，因为较大的那个现在已经是较小那个的父节点了。

图5-48中的`bubbleDown`函数实现了这一操作。在选择了与$A[i]$进行交换的子节点后，它会递归地调用自身，以消除新位置上的元素$A[i]$（也就是现在的`A[2i]`或`A[2i+1]`）与其新子节点之间可能存在的偏序树属性的破坏。参数n是堆中的元素数，或者说是最后一个元素的下标。

这个函数有点棘手。如果$A[i]$有两个子节点，就必须决定将其与哪个子节点交换，所以首先要在图5-48的第(1)行假设较大的子节点是$A[2i]$。而如果右子节点存在（即$child<n$），并且右子节点更大，第(2)行的测试就会得到满足，并在第(3)行让$child$成为$A[i]$的右子节点。

在第(4)行有两项需要测试的内容。首先，$A[i]$在该堆中有可能真的没有子节点。因此要通过检测是否有 $child \leqslant n$ 来确定$A[i]$是否为内部节点。第第二项测试是检测$A[i]$是否小于$A[child]$。如果两项条件都满足，那么在第(5)行就要将$A[i]$与它较大的那个子节点交换，并在第(6)行递归地调用`bubbleDown`，如果有必要的话，就将违背偏序树属性的元素进一步向下压。

```
     void bubbleDown(int A[], int i, int n)
     {
         int child;

(1)      child = 2*i;
(2)      if (child < n && A[child+1] > A[child])
(3)          ++child;
(4)      if (child <= n && A[i] < A[child]) {
(5)          swap(A, i, child);
(6)          bubbleDown(A, child, n);
         }
     }
```

图5-48　bubbleDown会将违背偏序树属性的元素压到合适的位置

可以按照图5-49所示的方式用bubbleDown实现优先级队列的deletemax操作。deletemax函数接受数组*A*和指向堆中当前元素数*n*的指针*pn*作为参数。这里省略了对 $n>0$ 的测试。

在第(1)行，将根节点处要删除的元素与最后的元素（在A[n]中）交换。技术上讲，应该返回删除的元素，不过，正如所看到的，将其放入不再属于该堆的A[n]也是可以的。

在第(2)行，将*n*减少1，实际上就是删除现在处于旧的A[n]中的最大元素。因为现在的根节点可能会违背偏序树属性，所以在第(3)行调用bubbleDown(A,1,n)，它会递归地将违背偏序树属性的元素向下压，直到该元素到达一个不再小于它子节点的位置或是成为叶子节点，不管哪种情况，都不再会违反偏序树属性了。

```
     void deletemax(int A[], int *pn)
     {
(1)      swap(A, 1, *pn);
(2)      --(*pn);
(3)      bubbleDown(A, 1, *pn);
     }
```

图5-49　用堆实现的优先级队列deletmax操作

✦ 示例 5.31

假设对图5-45中的堆执行deletemax操作。在交换了$A[1]$和$A[10]$之后，将*n*置为9。这样堆就变成了

1	2	3	4	5	6	7	8	9
5	18	16	9	7	1	9	3	7

在执行bubbleDown(A,1,9)时，会将*child*置为2。因为$A[2] \geqslant A[3]$，所以在图5-48的第(3)行不用增加*child*。然后，因为$child \leqslant n$而且$A[1] < A[2]$，所以交换这两个元素，得到数组

1	2	3	4	5	6	7	8	9
18	5	16	9	7	1	9	3	7

接着调用bubbleDown(A,2,9)。这要求我们在第(2)行对$A[4]$和$A[5]$加以比较，比较的结果是前者更大。因此，在图5-48的第(4)行，$child = 4$。又因为可知$A[2] < A[4]$，所以交换这两个元素，并对如下数组调用bubbleDown(A,4,9)。

1	2	3	4	5	6	7	8	9
18	9	16	5	7	1	9	3	7

再下来要比较$A[8]$和$A[9]$，结果后者更大，所以`bubbleDown(A,4,9)`的第(4)行有$child = 9$。因为$A[4] < A[9]$，所以再次进行交换，得到数组

1	2	3	4	5	6	7	8	9
18	9	16	7	7	1	9	3	5

随后调用`bubbleDown(A,9,9)`。在第(1)行将$child$置为18，而且因为$child < n$为假，第(2)行的第一个测试会失败。同样，第(4)行的测试也会失败，所以不用再进行交换或递归调用。这一数组现在已经是还原偏序树属性的堆了。

5.9.4 优先级队列操作的运行时间

优先级队列堆实现的每次插入或`deletemax`操作的运行时间是$O(\log n)$。要知道原因，首先考虑一下图5-47所示的`insert`程序。该程序前两步花的时间显然是$O(1)$，此外还要加上对`bubbleUp`的调用所花的时间。因此，需要确定`bubbleUp`的运行时间。

粗略地讲，我们注意到每次`bubbleUp`递归调用自身时，就是离根节点更近了一个节点的位置。因为平衡偏序树的高度大约是$\log_2 n$，所以递归调用的次数是$O(\log_2 n)$。因为对`bubbleUp`的每次调用花的时间是$O(1)$外加递归调用的时间（如果有的话），所以总时间应该为$O(\log n)$。

更严格地讲，设$T(i)$是`bubbleUp(A,i)`的运行时间，那么可以构建如下$T(i)$的递推关系。

依据。如果$i = 1$，那么$T(i)$是$O(1)$，因为很容易看出，在这种情况下，图5-46中的`bubbleUp`程序不会执行任何递归调用，只是执行了`if`语句的测试。

归纳。如果$i>1$，那么`if`语句的测试可能会失败，因为$A[i]$不再需要继续上升了。若该测试成功，则花$O(1)$的时间执行$swap$，并以参数$i/2$（若i为奇数则略小于$i/2$）递归调用`bubbleUp`一次。因此$T(i) \leqslant T(i/2) + O(1)$。

所以可以说，对于某些常数a和b，递推关系

$$T(1) = a$$
$$T(i) = T(i/2) + b,\ i > 1,$$

能作为`bubbleUp`运行时间的上界。如果将$T(i/2)$展开，就得到，对每个j，有

$$T(i) = T(i/2^j) + bj \tag{5.2}$$

如3.10节中那样，可以对j的值加以选择，使得$T(i/2^j)$最为简单。在这种情况下，可以令j等于$\log_2 i$，这样一来$i/2^j = 1$。因此，(5.2)式就成了$T(i) = a + b\log_2 i$，也就是说$T(i)$是$O(\log i)$。因为`bubbleUp`的运行时间是$O(\log i)$，所以插入操作的运行时间也是$O(\log i)$。

现在考虑$deletemax$。从图5-49中可以看出，$deletemax$的运行时间是$O(1)$加上`bubbleDown`的运行事件。对图5-48所示`bubbleDown`函数的分析基本上与对`bubbleUp`的分析相同。这里就不再赘述分析过程，直接得出`bubbleDown`和$deletemax$的运行时间也是$O(\log n)$这一结论。

5.9.5 习题

(1) 考虑图5-45中的堆，说明在下列情况下分别会发生什么。

(a) 插入3

(b) 插入20

(c) 删除最大元素

(d) 再次删除最大元素

(2) 通过对i的归纳证明(5.2)式。

(3) 通过对违背偏序树属性的节点的深度进行归纳，证明图5-46中的`bubbleUp`函数可以正确地将有一处违背偏序树属性的树还原为具有偏序树属性的树。

(4) 证明：如果`A`之前是大小为$n-1$的堆，那么`insert(A,x,n)`函数可以使`A`变为大小为n的堆。

(5) 通过对违背偏序树属性的节点的高度进行归纳，证明图5-48中的`bubbleDown`函数可以正确地将有一处违背偏序树属性的树还原为具有偏序树属性的树。

(6) 证明：`deletemax(A,n)`可以让大小为n的堆变为大小为$n-1$的堆。如果`A`之前不是堆，会发生什么？

(7) 证明：`bubbleDown(A,1,n)`处理长度为n的堆所花时间是$O(\log n)$。

(8) ** 随机选出不同优先级的n个元素构成堆，该堆是偏序树的概率是多少？如果没法总结出一般规则，就编写递归函数计算这一表示为n的函数的概率。

(9) 实现偏序树不一定要使用堆。假设使用之前用于二叉树的常规的左子节点右子节点数据结构。展示如何使用这一结构实现`bubbleDown`、`insert`和`deletemax`函数。

(10) * 二叉查找树也可以用作优先级队列的抽象实现。展示如何使用具有左子节点右子节点数据结构的二叉查找树实现插入和`deletemax`操作。这些操作在最坏情况下以及在一般情况下的运行时间分别是多少？

5.10　堆排序：利用平衡偏序树排序

现在要介绍被称为*堆排序*的算法。它会分两个阶段为数组`A[1..n]`排序。在第一个阶段，堆排序会给A偏序树属性。堆排序的第二个阶段会反复从堆中选出剩余元素中的最大元素，直到堆只由最小的元素构成，这样就完成了对数组A的排序。

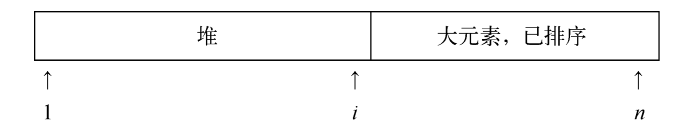


图5-50　堆排序过程中数组A的情况

图5-50展示了处于第二阶段的数组A。数组的开头部分具有偏序树属性，而剩下的部分则是以非递减次序排好序的元素。此外，已排序部分是数组中前$n-i$大的元素。在第二阶段，i的值可以从n减到1，从而让一开始为整个数组A的堆，最终减少到只剩下位于`A[1]`的最小元素。更详细地讲，第二阶段由如下步骤组成。

(1) 将`A[1..i]`中最大元素$A[1]$与$A[i]$交换。因为`A[i+1..n]`中所有元素都不小于`A[1..i]`中的元素，而且我们刚刚将`A[1..i]`中最大的元素移动到了位置i，所以可知`A[i..n]`是数组中前$n-i-1$大的元素，而且已经是排好次序的。

(2) i的值是递减的，每次将堆的大小减少1。

(3) 通过下压根节点处的元素（就是刚移动到$A[1]$的元素），还原开头部分的偏序树属性。

✦ 示例 5.32

考虑图5-45中的数组，它是具有偏序树属性的。这里从第二阶段的第一次迭代开始分析。在第一步中，要将$A[1]$与$A[10]$交换，得到：

1	2	3	4	5	6	7	8	9	10
5	18	16	9	7	1	9	3	7	18

第二步是将堆的大小减小为9，而第三步则是通过调用`bubbleDown(1)`还原前9个元素的偏序树属性。在这次调用中，A[1]和A[2]进行了交换。

1	2	3	4	5	6	7	8	9	10
18	5	16	9	7	1	9	3	7	18

接着，A[2]与A[4]进行了交换。

1	2	3	4	5	6	7	8	9	10
18	9	16	5	7	1	9	3	7	18

最后，A[4]和A[9]交换：

1	2	3	4	5	6	7	8	9	10
18	9	16	7	7	1	9	3	5	18

至此，`A[1..9]`具有了偏序树属性。

第二阶段的第二次迭代首先要交换A[1]中的元素18与A[9]中的元素5。在将5往下压到合适位置后，数组就成了

1	2	3	4	5	6	7	8	9	10
16	9	9	7	7	1	5	3	18	18

到这里，数组中最后两个元素已经是最大的两个元素，而且是已经排好次序的。

第二阶段会不断继续，直到完成对数组的排序。

1	2	3	4	5	6	7	8	9	10
1	3	5	7	7	9	9	16	18	18

5.10.1 数组的堆化

可以非正式地将堆排序描述为：

```
for (i = 1; i <= n; i++)
    insert(a_i);
for (i = 1; i <= n; i++)
    deletemax
```

要实现这一算法，先将待排序的n个元素a_1、a_2、…、a_n插入一个最初为空的堆中。然后执行n次`deletemax`操作，按从大到小的次序取出元素。图5-50所示的安排让我们可以随着数组中堆部分的萎缩，在数组的尾部存储已删除的元素。

我们已经在5.9节中论证过插入和`deletemax`操作的运行时间都是$O(\log n)$，而且每种操作显然都要执行n次，所以这是一种可与归并排序媲美的$O(n\log n)$排序算法。其实，在只需要最大的几个元素，而不需要整个已排序表的情况下，堆排序还能优于归并排序。原因在于，要让

数组变成堆，如果使用图5-51所示的heapify函数，只需要$O(n)$的时间就能完成，而不是$O(n\log n)$。

```
void heapify(int A[], int n)
{
    int i;

    for (i = n/2; i >= 1; i--)
        bubbleDown(A, i, n);
}
```

图5-51　数组的堆化

5.10.2　Heapify的运行时间

首先，图5-51中对bubbleDown的$n/2$次调用总时间看起来应该是$O(n\log n)$，因为我们了解的bubbleDown运行时间上界只有$\log n$这一个。不过，如果利用向下压元素的序列大多非常短这一事实，就可以得到更紧的边界——$O(n)$。

一开始，甚至都不必堆数组的后半部分调用bubbleDown，因为那里的节点全部是叶子节点。如果数组的第二个四分之一部分——也就是A[(n/4)+1..n/2]——中的元素存在比它们的子节点小的，就可以调用bubbleDown一次。不过，它们的子节点是在数组后半部分，都是叶子节点，因此，在A的第二个四分之一中，最多调用一次bubbleDown。同样，在数组的第二个八分之一中，最多调用两次bubbleDown。在数组个区域中调用bubbleDown的次数如图5-52所示。

	$n/16$	$n/8$	$n/4$	$n/2$	n
A	···	≤3	≤2	≤1	0

图5-52　随着数组下标不断变大，对bubbleDown的调用次数迅速减少

现在来计算一下heapify调用了多少次bubbleDown，其中包括递归调用。从图5-52可看出，可以将A分为若干个区段，其中第i个区段是由大于$n/2^{i+1}$且不大于$n/2^i$的j对应的A[j]组成。因此，区段i中的元素数就是$n/2^{i+1}$，而且区段i中每个元素至多调用i次bubbleDown。此外，$i>\log_2 n$的区段都为空，因为它们至多包含$n/2^{1+\log_2 n}=1/2$个元素。A[1]是区段$\log_2 n$中唯一的元素，因此需要计算和值

$$\sum_{i=1}^{\log_2 n} in/2^{i+1} \tag{5.3}$$

将(5.3)的有限和扩展为无限和，并提取出因式$n/2$，就可以给出该和值的上界

$$\frac{n}{2}\sum_{i=1}^{\infty} i/2^i \tag{5.4}$$

现在必须得出(5.4)式中和值的上界。$\sum_{i=1}^{\infty} i/2^i$可以写为

$$(1/2)+(1/4+1/4)+(1/8+1/8+1/8)+(1/16+1/16+1/16+1/16)+\cdots$$

可以将这些2的乘方的倒数写为如图5-53所示的三角形。每一行都是公比为$1/2$的无穷几何级数，而其和则是级数中第一项的两倍，正如图5-33右侧所示。各行之和又形成了另一个几何级数，而它的和是2。

$$
\begin{aligned}
1/2+1/4+1/8+1/16+\cdots &= 1 \\
1/4+1/8+1/16+\cdots &= 1/2 \\
1/8+1/16+\cdots &= 1/4 \\
1/16+\cdots &= 1/8 \\
\cdots &= \cdots
\end{aligned}
$$

图5-53 将 $\sum_{i=1}^{\infty} i/2^i$ 排列为三角和

这样一来，(5.4)的上界为 $(n/2)\times 2=n$ 。也就是说，在函数heapify中调用bubbleDown的次数不超过n。因为已经得出每次调用花费的时间为$O(1)$，不含任何递归调用，所以可以得出结论：heapify花的总时间为$O(n)$ 。

5.10.3 完整的堆排序算法

堆排序C语言程序如图5-54所示。它使用整数数组A[1..MAX]表示堆。待排序的元素被插入A[1..n]中。图5-54中函数声明的定义包含在5.9节和5.10节中。

```
    #include <stdio.h>

    #define MAX 100

    int A[MAX+1];

    void bubbleDown(int A[], int i, int n);
    void deletemax(int A[], int *pn);
    void heapify(int A[], int n);
    void heapsort(int A[], int n);
    void swap(int A[], int i, int j);

    main()
    {
        int i, n, x;

        n = 0;
        while (n < MAX && scanf("%d", &x) != EOF)
            A[++n] = x;
        heapsort(A, n);
        for (i = 1; i <= n; i++)
            printf("%d\n", A[i]);
    }

    void heapsort(int A[], int n)
    {
        int i;

(1)     heapify(A, n);
(2)     i = n;
(3)     while (i > 1)
(4)         deletemax(A, &i);
    }
```

图5-54 对数组进行堆排序

第(1)行调用heapify，它将待排序的n个元素变成一个堆。第(2)行将标记堆尾的i初始化为n。第(3)和第(4)行的循环将deletemax应用$n-1$次。我们应该重新审视图5-49中的代码，会

看到deletemax(A,i)会将当前堆中最大的元素（永远是A[1]）与$A[i]$交换。这样一来，i每次会减少1，所以堆的大小也会缩小1。在第(4)行被deletemax“删除”的元素现在成为数列已排序尾部的一部分。它不大于之前的尾部A[i+1..n]中的任何元素，而不小于仍在堆中的任意元素。因此，声明的属性得到保持，堆中的所有元素都先于尾部的所有元素。

5.10.4　堆排序的运行时间

刚刚已经确定了第(1)行中的heapify函数花的时间与n成比例。第(2)行显然花了$O(1)$的时间。因为第(3)行和第(4)行的循环每进行一次，i就减少1，所以循环要进行$n-1$次。第(4)行中对deletemax的调用花的时间是$O(\log n)$。因此，整个循环的总运行时间为$O(n\log n)$。这一时间主导了第(1)行和第(2)行的运行时间，所以heapsort函数处理n个元素的运行时间是$O(n\log n)$。

5.10.5　习题

(1) 对3、1、4、1、5、9、2、6、5这列元素应用堆排序。

(2) * 给出一个运行时间是$O(n)$的算法，使其从具有n个元素的表中找出前$\sqrt{n}$大的元素。

5.11　小结

读者应该从本章中获取如下要点。

- 树是一种用于表示层次化信息的重要数据模型。
- 多种涉及数组和指针结合的数据结构可用于实现树，选择何种数据结构取决于要对树进行哪种操作。
- 树节点最重要的两种表示分别是最左子节点右兄弟节点表示和单词查找树（指向子节点的指针数组）。
- 递归算法和证明也适用于树。结构归纳法是普通归纳模式的一种变形，可以有效地对树中的节点数进行完全归纳。
- 二叉树是树模型的一种变形，它的每个节点最多可以有左子节点和右子节点。
- 二叉查找树是带标号的二叉树，它具有“二叉查找树属性”，即节点左子树的所有标号都先于该节点的标号，而且节点右子树的所有标号都后于该节点的标号。
- 词典抽象数据类型是可以对其执行插入、删除和查找操作的集合。二叉查找树可以有效地实现词典。
- 优先级队列是另一种抽象数据类型，是可以对其执行插入和deletemax操作的集合。
- 偏序树是种带标号的二叉树，它具有任意节点的标号都不小于其子节点标号的属性。
- 平衡偏序树除最下层之外的各层都被节点占满，而最下层只有靠左侧的位置被占据，它可以通过被称为堆的数组结构实现。这一结构提供了一种复杂度为$O(\log n)$的优先级队列实现，并带来了一种复杂度为$O(n\log n)$的排序算法——堆排序。

5.12　参考文献

树的单词查找树表示来自Fredkin [1960]。二叉查找树是由若干人各自独立发明的，而读者

可以参考Knuth [1973]来了解二叉查找树的历史以及大量与各类查找树有关的信息。要了解树的更多高级应用，参见Tarjan [1983]。

Williams [1964]率先设计了平衡偏序树的堆实现。Floyd [1964] 描述了堆排序的一个高效版本。

Floyd, R. W. [1964]. “Algorithm 245: Treesort 3,” *Comm. ACM* 7:12, pp. 701.

Fredkin, E. [1960]. “Trie memory,” *Comm. ACM* 3:4, pp. 490–500.

Knuth, D. E. [1973]. *The Art of Computer Programming*, Vol. III, *Sorting and Searching*, 2nd ed., Addison-Wesley, Reading, Mass.

Tarjan, R. E. [1983]. *Data Structures and Network Algorithms*, SIAM Press, Philadelphia.

Williams, J. W. J. [1964]. “Algorithm 232: Heapsort,” *Comm. ACM* 7:6, pp. 347–348.

第 6 章

表数据模型

和树一样，表也是计算机程序中最基础的数据模型之一。从某种意义上讲，表就是树的简化形式，因为大家可以将表视为每个左子节点都是叶子节点的二叉树。不过，表还能表示其他一些方面，这些方面与我们之前了解的关于树的那些情况不同。例如，我们将要谈论对表的操作，比如压入和弹出，这是没法用树来模拟的；而且要探讨字符串，这种特殊而重要的表需要它们自己的数据结构。

6.1 本章主要内容

6.2节介绍了与表有关的术语。本章其余部分将介绍以下主题。

- 表的基本操作（6.3节）。
- 由链表数据结构（6.4节）和数组数据结构（6.5节）实现的抽象表。
- 栈：只能从一端插入和删除的表（6.6节）。
- 队列：从一端插入从另一端删除的表（6.8节）。
- 字符串和用来表示字符串的特殊数据结构（6.10节）。

此外，我们还将详细研究表的两类应用。

- 运行时栈，C语言以及其他多种语言用来实现递归函数的方法（6.7节）。
- 找出两个字符串最长公共子序列的问题，及其通过“动态规划”（或者说填表）算法得出的解决方案（6.9节）。

6.2 基本术语

*表*是由0个或多个元素组成的有限序列。如果这些元素全是T类型的，那么就说该类型的表是“T表”。因此，就有整数表、实数表、结构体表、整数表的表，等等。一般可以预期列表的元素都是某一类型的。不过，因为一种类型可以是多种类型的联合，所以单一“类型”的限制是可以绕开的。

表通常表示为用逗号分隔表中各元素，并用一对圆括号将这些元素括起来，如

$$(a_1, a_2, \cdots, a_n)$$

其中a_i都是表中的元素。

在某些情况下，我们将不会把逗号和括号写出来。特别要说的是，我们将要研究字符串，

也就是由字符组成的表。字符串一般不会写上逗号或其他分隔符号，而且不会用括号括起来。字符串的元素通常都是用等宽字体表示的。`foo`就是由3个字符组成的表，其中第一个字符为`f`，而第二和第三个字符都是`o`。

✦ 示例 6.1

下面是一些表的例子。

(1) 小于20的质数按照从小到大的顺序组成的表：

(2，3，5，7，11，13，17，19)

(2) 稀有气体元素按照原子量从小到大的顺序排列组成的表：

(氦，氖，氩，氪，氙，氡)

(3) 平年各月天数组成的表：

(31，28，31，30，31，30，31，31，30，31，30，31)

这个例子提醒我们，同一元素可以在某个表中出现多次。

✦ 示例 6.2

一行文本是表的另一个例子。组成这行文本的单个字符就是表中的元素，所以该表就是个字符串。该字符串通常会包含若干空字符，而且一行文本的最后一个字符通常是“换行”符。

再举一个例子，文档也可视作表。这种情况下，表中的元素就是文本行。因此，文档就是由表作为元素组成的表，具体说来这些作为元素的表都是字符串。

✦ 示例 6.3

n维空间中的点可以表示为由n个实数构成的表。例如，单位正方体的顶点可以表示为图6-1所示的三元组。各表中的3个元素表示作为正方体8个角（“顶点”）之一的点的坐标。第一个元素表示x坐标（水平方向），第二个表示y坐标（向页内方向），第三个表示z坐标（垂直方向）。

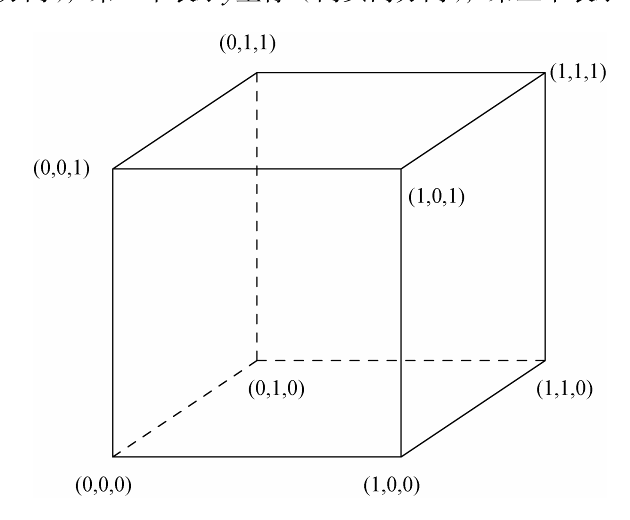


图6-1 表示为三元组的单位正方体顶点

6.2.1 表的长度

表的长度是指元素在表中出现的次数。如果表中元素数为0，那么就说该表为空表。我们用

希腊字母ϵ（音“伊普西龙”）表示空表。还可以用一对不包含任何内容的圆括号()表示空表。谨记，长度是按位置计算而不是按不同符号计算的，所以在表中出现k次的某一符号可以为表增加长度k。

✦ 示例 6.4

示例6.1中表(1)的长度是8，而表(2)的长度是6。表(3)的长度为12，因为每个月都要占据一个位置。而表中只有3个元素这一事实对表的长度来说是无关紧要的。

6.2.2 表的部分

如果表非空，那么它就是由称为表头（head）的第一个元素，以及称为尾部（tail）的表中其余部分组成的。例如，示例6.1中表(2)的表头就是“氦”，而该表的尾部则由其余5个元素组成：

(氖，氩，氪，氙，氡)

元素和长度为1的表

谨记，表的表头是个元素，而表的尾部却是表。此外，我们不应该将表的表头（假如为a）与只包含一个元素a的长度为1的表（通常会写为带括号的(a)）混淆。如果元素a是T类型的，那么表(a)就是“T表”类型的。

如果不能认识到这种区别，就可能在用数据结构实现表时造成编程错误。例如，我们可以用互相链接的单元表示表，这些单元通常是结构体，具有存放T类型元素的`element`字段，以及存放指向下一单元的指针的`next`字段。那么元素a就是T类型的，而表(a)则是具有存放着a的`element`字段和存放着`NULL`的`next`字段的单元。

如果有表$L=(a_1, a_2, \cdots, a_n)$，则对满足$1 \leqslant i \leqslant j \leqslant n$的$i$和$j$来说，$(a_i, a_{i+1}, \cdots, a_j)$是$L$的子表。也就是说，子表是由从某个位置$i$开始到某个位置$j$结束的所有元素组成的。还可以说空表$\epsilon$是任何表的子表。

表$L=(a_1, a_2, \cdots, a_n)$的子序列是指从$L$中剔除0个或多个元素后形成的表。剩下的这些元素，也就是构成子序列的这些元素，必须按照与出现在L中相同的顺序排列，不过子序列的元素在L中不一定是连续的。请注意，ϵ和表L本身总是L的子序列，而且L的子表也是L的子序列。

✦ 示例 6.5

设L是字符串`abc`，那么L的子表有

ϵ，a，b，c，ab，bc，abc

它们也都是L的子序列。除此之外，`ac`也是子序列，但它不是子表。

再举个例子，设L是字符串`abab`，那么子表就有

ϵ，a，b，ab，ba，aba，bab，abab

这些也是L的子序列。除此之外，L还有子序列`aa`，`bb`，`aab`，`abb`。请注意，`bba`这样的字符串不是L的子序列。即便L中确实有两个b和一个a，但它们在L中出现的顺序，不能让我们通过从L中剔除某些元素构成`bba`。也就是说，在L中，第二个b后面是没有a的。

表前缀是指从表的开头开始的任意子表。表后缀则是以表的结尾为末尾的子表。空表ϵ是种特殊情况，它是任意表的前缀和后缀。

✦ 示例 6.6

表的前缀有ϵ，a，ab和abc，而它的后缀是ϵ，c，bc和abc。

Car和Cdr

在Lisp语言中，表头叫作*car*，而尾部则称为*cdr*（音“cudder”）。术语“*car*”和“*cdr*”的名字来源于IBM 709型计算机中机器指令的两个字段，Lisp最早就是在该型计算机上实现的。car表示“contents of the address register”（地址寄存器的内容），而cdr则表示“contents of the decrement register”（减量寄存器的内容）。某种意义上讲，存储字（memory word）可以视为有着element和next字段（分别对应car和cdr）的单元。

6.2.3 表中元素的位置

表中的每个元素都有与之关联的位置。如果有表$(a_1, a_2, \cdots, a_n)$，而且$n \geqslant 1$，a_1就是第一个元素，a_2就是第二个元素，以此类推，而a_n则是最后一个元素。还可以说a_i出现在位置i。除此之外，a_i是在a_{i-1}之后，在a_{i+1}之前。而存放元素a的位置则称作a的出现。

表中位置的数量就等于表的长度。同一元素是有可能出现在两个或多个位置的，因此不要把位置和出现在该位置的元素弄混了。例如，示例6.1中的表(3)就有12个位置，其中有7个存放着31，分别是位置1、3、5、7、8、10和12。

6.2.4 习题

(1) 针对表(2，7，1，8，2)回答以下问题。
  (a) 它的长度是多少？
  (b) 它的前缀有哪些？
  (c) 它的后缀有哪些？
  (d) 它的子表有哪些？
  (e) 它有多少个子序列？
  (f) 它的表头是什么？
  (g) 它的尾部是什么？
  (h) 它包含多少个位置？

(2) 对字符串banana重复习题(1)中的练习。

(3) ** 在长度为$n \geqslant 0$的表中，最多可能有多少(a)前缀；(b)子表；(c)子序列？而最少又分别可能有多少？

(4) 如果表L尾部的尾部是空表，那么L的长度为多少？

(5) * 胡图写了个由整数表组成的表，不过他省略了括号，结果成了：1，2，3。而这可以表示很多由表组成的表，比如$\big((1),(2,3)\big)$。那么在不含空表作为元素的情况下，所有可能的表都有哪些？

6.3 对表的操作

可以对表执行多种不同的操作。第2章中，当我们讨论归并排序时，基本问题就是为表排序，不过我们还需要将表一分为二，再合并两个已排序的表。从形式上讲，为表$(a_1, a_2, \cdots, a_n)$排序的操作就是将表替换为由其元素的排列组成的表$(b_1, b_2, \cdots, b_n)$，其中$b_1 \leqslant b_2 \leqslant \cdots \leqslant b_n$。这里就

像之前一样，≤表示元素的次序关系，比如整数或实数的“小于或等于”，或是字符串的词典次序。合并两个已排序表的操作是用给定的两个表构建一个包含其中相同元素的已排序表。多重性必须得到保持，也就是说，如果某个元素a在给定的两个表中出现了k次，那么得出的表中a也出现k次。回顾2.8节就能看到对表的这两种操作的示例。

6.3.1　插入、删除和查找

回想一下5.7节，“词典”是可以对其执行插入、删除和查找操作的元素集合。集合与表之间存在一个重要区别。虽然元素在集合中绝不能出现多次，但正如我们所见，元素在表中可出现多次。集合的问题将在第7章中讨论。不过，表可以实现集合，其方式是将集合$\{a_1, a_2, \cdots, a_n\}$中的元素以任意次序放置到表中，例如次序$(a_1, a_2, \cdots, a_n)$，或次序$(a_n, a_{n-1}, \cdots, a_1)$。因此，如果一些对表的操作与对集合的词典操作类似，应该不会让人感到奇怪。

(1) 可以向表L中插入元素x。从原则上讲，x可能出现在表中任何位置，而且x在L中出现一次或多次都是没关系的。我们通过在表中增加一次x的出现来插入x。作为一种特例，如果让x作为新表的表头（这样一来L就成了尾部），就是将x压入表L中。如果$L=(a_1, a_2, \cdots, a_n)$，那么得到的表就是$(x, a_1, a_2, \cdots, a_n)$。

(2) 可以从表L中删除元素x。这里，是从L中删除x的一次出现。如果x出现多次，那么就要指出删除哪个x。例如，我们可以总是删除第一个x。如果想要删除所有的x，就要重复删除操作，直到不再剩下x为止。如果表L中未出现x，那么删除操作就不会造成任何影响。作为特例，如果是删除了表的表头元素，使表$(x, a_1, a_2, \cdots, a_n)$变成了$(a_1, a_2, \cdots, a_n)$，就说是弹出表。

(3) 可以在表L中查找元素x。这一操作会返回`TRUE`或`FALSE`，具体取决于x是否为表中元素。

✦　示例 6.7

设L为表(1，2，3，2)。如果我们选择压入1，也就是将1插入到表头位置，则$insert(1，L)$的结果是(1，1，2，3，2)。而若是将1插入到末尾，就得到(1，2，3，2，1)。此外，这个新的1还可以被放置到表L内部3个位置中的任何一个上。

如果删除表中第一个2，那么$deletemax(2，L)$的结果是表(1，3，2)。若问$lookup(x，L)$，则当x为1、2或3时，答案为`TRUE`，而当x为其他值时，答案为`FALSE`。

6.3.2　串接

要串接表L和表M，就是以L的元素作为开头部分，后面接上M的元素形成新表。也就是说，如果$L=(a_1, a_2, \cdots, a_n)$，而$M=(b_1, b_2, \cdots, b_k)$，那么$L$和$M$的串接$LM$就是表

$$(a_1, a_2, \cdots, a_n, b_1, b_2, \cdots, b_k)$$

请注意，空表是串接恒等的。也就是说，对任何表L都有$\epsilon L = L\epsilon = L$。

✦　示例 6.8

如果L是表(1，2，3)，M是表(3，1)，那LM就是表(1，2，3，3，1)。如果L是字符串`dog`，而M是字符串`house`，那LM就是字符串`doghouse`。

6.3.3　表的其他操作

另一类对表的操作是与表的特定位置相关的。例如

(a) $first(L)$会返回表L的第一个元素（表头），而$first(L)$会返回L的最后一个元素。如果L是空表，则这两种操作都会导致错误出现。

(b) $retrieve(i，L)$操作会返回表L中第i个位置处的元素。如果L的长度小于i，就会出错。

除此之外，还有涉及表的长度的操作。常见的包括下列两种。

(c) $length(L)$，返回表L的长度。

(d) $isEmpty(L)$，如果L为空表则返回`TRUE`，否则返回`FALSE`。而$isNotEmpty(L)$会返回相反的结果。

6.3.4　习题

(1) 设L是表(3，1，4，1，5，9)，回答下列问题。

(a) $delete(5，L)$的值是多少？

(b) $delete(1，L)$的值是多少？

(c) 弹出L的结果是什么？

(d) 将2压入表L的结果是什么？

(e) 如果以元素6和表L执行$lookup$，会返回什么？

(f) 如果M是表(6，7，8)，那么LM（L和M的串接）的值是多少？ML的值又是多少？

(g) $first(L)$是多少？$last(L)$又是多少？

(h) $retrieve(3，L)$的结果是多少？

(i) $flengh(L)$的值是多少？

(j) $isEmpty(L)$的值是多少？

(2) ** 如果L和M是表，在什么条件下有$LM=ML$？

(3) ** 设x是元素而L是表，那么在什么条件下以下等式为真？

(a) $delete\left(x，insert(x，L)\right)=L$

(b) $insert\left(x，delete(x，L)\right)=L$

(c) $first(L)=retrieve(1，L)$

(d) $last(L)=retrieve\left(length(L)，L\right)$

6.4　链表数据结构

实现表的最简单方式就是使用链表。每个链表单元都由两个字段构成，一个字段包含着表中的元素，另一个字段则含有指向链表下一单元的指针。简单起见，假设元素都是整数。我们不仅能使用具体的`int`类型来表示元素的类型，而且能用`==`、`<`等标准比较运算符来比较元素。本节习题将会启发读者编写这些函数的变体，使其能处理任意类型的元素，而元素的比较则是由用户定义的函数进行的，比如测试相等性的eq，及测试x在次序上是否先于y的$lt(x,y)$，等等。

接下来，要使用来自1.6节中的宏：

```
DefCell(int, CELL, LIST);
```

它可展开为表示单元和表的标准结构体

```
typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
};
```

请注意，LIST是指向单元的指针类型。实际上，每个单元的next字段既指向下一个单元，也指向表中剩余的所有部分。

图6-2展示了表示抽象表 $L=(a_1, a_2, \cdots, a_n)$ 的链表。每个单元都对应一个元素，元素a_i出现在第i个单元的element字段。对$i = 1, 2, \cdots, n-1$而言，第i个单元中的指针是指向第i+1个单元的，而最后一个单元中的指针为NULL，表示这是表的末尾。在表之外是个名为L的指针，它指向该表的第一个单元，L是LIST类型的。如果表L为空，L的值就为NULL。

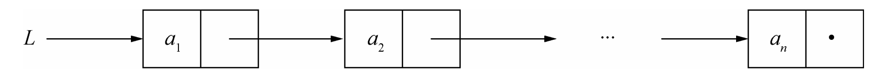


图6-2　表示表 $L=(a_1, a_2, \cdots, a_n)$ 的链表

表和链表

请记住，表是一种抽象模型，或者说是数学模型。而链表则是种简单的数据结构，这在第1章中提到过。虽然链表是实现表数据模型的一种方式，但正如我们所见，它并非实现表数据模型的唯一方式。无论如何，这是再次记住模型与实现模型的数据结构之间区别的良好时机。

6.4.1　词典操作的链表实现

如果用链表表示词典，那么该如何实现操作？以下对词典的操作是在5.7节中定义的。

(1) *insert* (x，D)，将元素x插入词典D中；

(2) *delete* (x，D)，从词典D中删除元素x；

(3) *lookup* (x，D)，确定元素x是否在词典D中。

我们将看到，与之前章节中讨论过的二叉查找树相比，链表是一种更为简单的实现词典的数据结构。不过，在使用链表表示时，词典操作的运行时间不像使用二叉查找树时那么少。在第7章中还将看到一种更佳的表示词典的方式——散列表，它利用对表的词典操作作为子例程。

这里假设我们的词典包含的是整数，而且单元是按照本节开头那样定义的。那么词典的类型就是LIST，也是像本节开头那样定义的。含有元素集合 $\{a_1, a_2, \cdots, a_n\}$ 的词典可以用图6-2中的链表表示。还有很多其他的表可以表示这一集合，因为元素的次序在集合中是无关紧要的。

6.4.2　查找

要执行*lookup*(x，D)，就要对表示D的表中的每个单元加以检验，看看它是否存放了所需的元素x。如果是，就返回TRUE。如果到达表末仍未发现x，就返回FALSE。一如之前那样，定义的常量TRUE和FALSE表示常数1和0，BOOLEAN则表示定义的类型int。递归函数lookup(x,D)如图6-3所示。

如果表的长度为n，就说图6-3中的函数所花的时间为 $O(n)$ 。除了结尾的递归调用外，lookup花的时间是 $O(1)$ 。当调用执行之后，剩余的表的长度要比表L的长度小1。因此对长度为n的表执行lookup要花上 $O(n)$ 的时间应该不会让人感到意外。更加正式地讲，以下递推关系给出了当第二个参数指向的表L长度为n时lookup的运行时间。

```
BOOLEAN lookup(int x, LIST L)
{
    if (L == NULL)
        return FALSE;
    else if (x == L->element)
        return TRUE;
    else
        return lookup(x, L->next);
}
```

图6-3　在链表中查找

依据。$T(0)=O(1)$，因为当L为`NULL`时，没有进行递归调用。

归纳。$T(n)=T(n-1)+O(1)$。

正如我们在第3章中见过很多次，这一递推关系的解是$T(n)=O(n)$。因为含n个元素的词典是用长度为n的表表示的，所以对大小为n的词典执行查找操作所花的时间也是$O(n)$。

不幸的是，进行一次成功查找的平均时间也与n成比例。如果要查找的元素x确定在D中，那么x在表中位置的期望值为$(n+1)/2$。也就是说，x会等可能地出现在从第一个元素到第n个元素中的任一位置。因此，递归调用`lookup`的次数的期望值是$(n+1)/2$。因为每次调用所花的时间是$O(1)$，所以平均成功查找所花的时间为$O(n)$。当然，如果查找不成功，那么在到达表末并返回`FALSE`之前，已经进行了全部n次调用。

6.4.3　删除

从链表中删除元素x的函数如图6-4所示。第二个参数`pL`是指向表L的指针，而不是表L本身。这里使用了“按引用调用”的风格，因为我们希望`delete`可以从表中删除含有x的单元。随着我们沿着表向下移动，`pL`中存放着一个指针，它指向的是指向“当前”单元的指针。如果在第(2)行发现x在当前单元C中，就接着在第(3)行改变指向单元C的指针，使得它指向该表中紧跟在C之后的那个单元。如果C正好在表的末尾，之前指向C的指针就成了`NULL`。如果x不是当前的元素，那么在第(4)行就递归地从表尾删除x。

请注意，如果表为空表，那么第(1)行的测试会使该函数在没有任何动作的情况下返回。这是因为x不会出现在空表中，而我们不需要采取任何措施来从词典中删除x。如果D是表示词典的链表，那么调用`delete(x,&D)`就会初始化从词典D中删除x的操作。

```
    void delete(int x, LIST *pL)
    {
(1)     if ((*pL) != NULL)
(2)         if (x == (*pL)->element)
(3)             (*pL) = (*pL)->next;
            else
(4)             delete(x, &((*pL)->next));
    }
```

图6-4　删除元素

如果元素x没有出现在表示词典D的链表中，那么就会继续向下运行直到表的末端，为每个元素花上$O(1)$的时间。分析过程类似对图6-3中`lookup`函数的分析，这里就将细节留给读者自己来分析吧。因此，如果D有n个元素，那么删除不在D中的元素所花的时间是$O(n)$。如果x在

词典D中，那么平均下来，将会在表中接近中点的位置遇到x。因此，平均要查找$(n+1)/2$个单元，而成功删除操作的运行时间也是$O(n)$。

6.4.4　插入

向链表中插入元素x的函数如图6-5所示。要插入x，需要确定x没有出现在表中。如果它已经在表中，就什么都不用做。如果x未出现，就必须将其添加到表中。将x添加到表中的什么位置并不重要，不过图6-5中的函数是将x添加到表的末尾。在第(1)行检测到末尾的`NULL`时，就确定x不在表中。那么，第(2)到第(4)行就会将x添加到表尾。

如果表非`NULL`，第(5)行就会检查x是否在当前单元。如果x不在这里，第(6)行就会对表的尾部进行递归调用。如果在第(5)行就找到x，那么函数`insert`就会终止，不进行任何递归调用，而且不会对表L造成任何改变。调用`insert(x,&D)`会初始化将x插入词典D的操作。

```
     void insert(int x, LIST *pL)
     {
(1)      if ((*pL) == NULL) {
(2)          (*pL) = (LIST) malloc(sizeof(struct CELL));
(3)          (*pL)->element = x;
(4)          (*pL)->next = NULL;
         }
(5)      else if (x != (*pL)->element)
(6)          insert(x, &((*pL)->next));
     }
```

图6-5　元素的插入

与查找和删除的情况一样，如果在表中没有找到x，就会到达表的末端，花费$O(n)$的时间。如果找到x，那么平均会走过表中一半[①]的位置，而且平均而言仍会花$O(n)$的时间。

6.4.5　带重复的插入、查找和删除

如果在执行插入操作之前不检查x是否出现在表中，可以让插入操作运行得更快。不过，这样做的后果就是，在表示词典的表中可能有某一元素的多个副本。

要执行词典操作$insert(x, D)$，只要创建一个新单元，将x放进去，并将该单元压入表示D的表的开头即可。这一操作花费$O(1)$的时间。

查找操作就和图6-3中所示的一模一样。唯一的不便就是可能要查找更长的表，因为表示词典D的表的长度可能会大于D中的成员数。

重提抽象和实现

大家可能会感到惊讶，我们在表示词典的表中使用了重复，因为抽象数据类型DICTIONARY被定义为集合，而集合是不含重复的。不过，有重复的并不是词典，而实现词典的数据结构可以有重复。但是，即便当x在链表中多次现身，它在链表表示的词典中也只会出现一次。

---

① 在后面的分析中，当说到长度为n的表的中点时，将会使用“一半”或“$n/2$”这样的说法。严格地说，$(n+1)/2$要更加精确。

删除操作稍有区别。在遇到含有元素x的单元时，我们不能停止对x的查找，因为表中可能还有x的其他副本。因此，就算表L的表头包含x，也必须将x从L的尾部删除。这样一来，我们不只要处理更长的表，而且要实现成功的删除操作，就必须查找每个单元，而不像表中不允许出现重复的情况时那样是平均查找表的一半。这些带重复的词典操作的细节就留作本节习题了。

总之，通过允许重复，可以让插入操作变得更快，时间是 $O(1)$ 而不是 $O(n)$ 。不过，成功的删除操作需要对整个表进行查找，而不是平均查找一半列表。而且对于查找和删除，必须要处理比不允许重复时更长的表，虽然长多少取决于插入词典中已存在元素的频率有多高。

要选择哪种方法是有点技巧的。显然，如果插入操作占主导地位，就应该允许重复。在极端情况下，如果只插入而从不查找或删除，就能让每次操作具有 $O(1)$（而不是$O(n)$）的性能。[①] 如果有理由确定从不会插入词典中已存在的元素，就可以使用快速插入和快速删除，那样只要找到待删除元素的一次出现就可以停下了。另一方面，如果有可能插入重复的元素，而且查找或删除又占主导地位，那么在插入x之前最好还是先检查一下它是否已经存在于表中，就如图6-5所示的insert函数那样。

6.4.6 表示词典的已排序表

另一种方案是让表示词典D的表中的元素一直按照递增次序排好序。然后，如果希望查找元素x，只要行进到x可能出现的位置就行了，平均而言，也就是表的中间位置。如果遇到大于x的元素，就说明在后面的部分没希望找到x了。因此就不用沿着表行进以继续进行失败的查找了。这样做可以节省为2的因数（开销变为一半），不过确切的因数是有些模糊的，因为在表中每遇到一个元素就必须询问按照排序次序x是否在其之后。不过，在进行插入和删除操作时，在不成功的查找上也能节约同样的开销。

用于已排序表的查找函数如图6-6所示。把图6-4和图6-5所示函数修改为处理已排序表的版本的工作就留作本节习题了。

```
BOOLEAN lookup(int x, LIST L)
{
    if (L == NULL)
        return FALSE;
    else if (x > L->element)
        return lookup(x, L->next);
    else if (x == L->element)
        return TRUE;
    else /* 这里有 x < L->element,
             因此x不可能在已排序表 L 中*/
        return FALSE;
}
```

图6-6 在已排序表中查找

6.4.7 各种方法的比较

图6-7中的表格表明了对我们讨论过的3种基于表的词典表示，执行3种词典操作各自必须查找的单元数。设词典中有n个元素，如果不允许重复，这也就是表示词典的表的长度。在允许重

① 如果从来都不管词典中有什么，还干嘛费事往里面插东西呢？

复出现时，我们用m来表示该表的长度。我们知道$m \geqslant n$，但不知道m比n大多少。在用到$n/2 \to n$这样的表示方式时，意思是当查找成功时，平均查找了$n/2$个单元，而在不成功时，平均查找了n个单元。而$n/2 \to m$这样的项则表示，在一次成功的查找中，我们在看到要查找的元素之前，平均会看到词典中$n/2$个元素[①]，但在失败的查找中，就必须走完整个长度为m的表，直到到达其末端。

	插　入	删　除	查　找
无重复	$n/2 \to n$	$n/2 \to n$	$n/2 \to n$
有重复	0	m	$n/2 \to m$
已排序	$n/2$	$n/2$	$n/2$

图6-7　3种用链表表示词典的方法所查找的单元数

请注意，除了有重复情况下的插入操作外，这些运行时间都要比数据结构为二叉查找树时词典操作的平均运行时间长。正如我们在5.8节中所见，在使用二叉查找树时，词典操作平均所花时间为$O(\log n)$。

明智的测试次序

请注意图6-6的程序中3项测试的次序。首先要测试L不为NULL。我们没有其他的选择，因为如果L为NULL，则其余两项测试会导致错误。设y是L->element的值。那么除了最后一个单元外，在每个访问过的单元都有$x>y$。因为如果有$x=y$，就是成功完成了查找，而如果$x<y$，就是没能找到x而终止。因为首先要测试$x>y$，而且当且仅当它失败时，我们才需要区分另两种情况。测试的这种次序遵循这样一个基本原则：要首先测试最平常的情况，并因此节约平均要执行的总测试数。

如果访问了k个单元，就要测试k次L是否为NULL，而且要测试k次x是否大于y。并且还要测试一次是否有$x=y$，这样总共要进行$2k+1$次测试。也就是说，只比图6-3中利用未排序表的lookup函数成功找到元素x的情况多一次测试。如果未找到该元素，可以预期在图6-6中使用的测试要比在图6-3中使用的测试少得多，因为图6-6中平均只要检查一半单元后就会停止。因此，虽然不管使用已排序表还是使用非排序表，词典操作的大O运行时间都是$O(n)$，但是如果使用已排序表的话，通常会有常数因子上的轻微优势。

6.4.8　双向链表

在链表中，要从某个单元向表的开头移动不是那么容易的。而双向链表是这样一种数据结构，它让表中向前和向后的移动都非常方便。整数双向链表中的单元包含3个字段：

```
typdef struct CELL *LIST;
struct CELL {
    LIST previous;
    int element;
    LIST next;
};
```

① 事实上，因为可能存在重复，所以在看到$n/2$个不同元素前，可能已经检查了多于$n/2$个元素。

多出来的这个字段含有指向表中前一个单元的指针。图6-8展示了表示表 $L=(a_1, a_2, \cdots, a_n)$ 的双向链表数据结构。

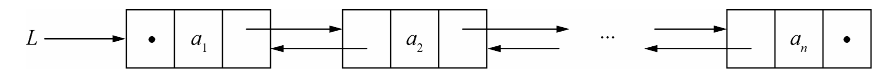


图6-8 表示表 $L=(a_1, a_2, \cdots, a_n)$ 的双向链表

双向列表结构上的词典操作基本与单向链表上的那些操作相同。要了解双向链表的优势，可以考虑只给定指向元素a_i所在单元的指针时删除该元素的操作。在单向链表中，我们要通过从头开始查找该表来找出前一个单元。而有了双向链表，就可以通过如图6-9所示的一系列指针操作，在$O(1)$时间里完成这一操作。

```
      void delete(LIST p, LIST *pL)
      {
          /* p是指向待删除单元的指针，
             而pL是指向链表的指针 */
(1)       if (p->next != NULL)
(2)           p->next->previous = p->previous;
(3)       if (p->previous == NULL) /* p 指向第一个单元 */
(4)           (*pL) = p->next;
          else
(5)           p->previous->next = p->next;
      }
```

图6-9 从双向链表中删除元素

图6-9中所示的`delete(p,pL)`函数接受指向待删除单元的指针p，以及指向表L本身的指针pL作为参数。也就是说，pL是指向表中第一个单元的指针的地址。在图6-9的第(1)行中，我们要检查p有没有指向最后一个单元。如果没有的话，那么在第(2)行，我们会让接下来那个单元的反向指针指向在p之前的那个单元。如果p正好指向第一个单元的话，就让它等于`NULL`。

第(3)行会测试p是否为第一个单元。如果是，那么在第(4)行我们会让pL指向第二个单元。请注意，在这种情况下，第(2)行会让第二个单元的`previous`字段变为`NULL`。如果p不是指向第一个单元，那么在第(5)行我们会让前一个单元的正向指针指向p之后的那个单元。这样一来，由p指向的那个单元就顺利地与表分离了，其前一个单元和后一个单元现在是互相指向的。

6.4.9 习题

(1) 为(a)图6-4中`delete`函数，(b)图6-5中`insert`函数的运行时间建立递推关系。它们的解各是多少？

(2) 为使用带重复链表的词典操作插入、查找和删除编写C语言函数。

(3) 为如图6-6那样使用已排序表的插入和删除操作编写C语言函数。

(4) 编写C语言函数，使其能在双向链表中由p指向的单元之后的新单元中插入元素x。图6-9是用于删除的相似函数，不过对插入操作来说，我们不需要知道表头L。

(5) 如果使用双向链表数据结构，一种选择是不通过指向单元的指针表示表，而通过具有未使用`element`字段的单元来表示。请注意，这一“表头”单元本身并非表的一部分。该“表头”的`next`字段指向该表真正的第一个单元，而这第一个单元的`previous`字段则指向该“表头”单元。然后可以在不知道表头L（正是我们在图6-9中需要知道的）的情况下删除由指针p指向的单元，而

不是那个未使用element字段的“表头”。编写C语言函数，使其利用这里描述的格式从双向链表中删除元素。

(6) 编写递归函数，实现使用链表数据结构的(a) $retrieve(i, L)$；(b) $length(L)$；(c) $last(L)$。

(7) 扩展下列函数，使其单元可以接受任意类型ETYPE的元素，使用函数 $eq(x, y)$ 测试x和y是否相等，并用 $lt(x, y)$ 分辨x是否在ETYPE类型元素的次序下先于y。

(a) 图6-3中的*lookup*。

(b) 图6-4中的*delete*。

(c) 图6-5中的*insert*。

(d) 使用带重复表的*insert*、*delete*和*lookup*。

(e) 使用已排序表的*insert*、*delete*和*lookup*。

6.5　表基于数组的实现

实现表的另一种常见方式是创建由下列两部分组成的结构体。

(1) 存放元素的数组；

(2) 记录表中当前元素数量的变量length。

图6-10展示了如何使用数组A[0..MAX-1]表示表 $(a_0, a_1, \cdots, a_{n-1})$。元素$a_0$、$a_1$、…、$a_{n-1}$存储在A[0..n-1]中，而且 $length = n$。

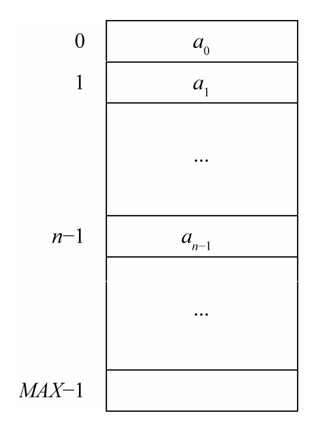


图6-10　存放表 $(a_0, a_1, \cdots, a_{n-1})$ 的数组A

就像在6.4节中那样，我们假设表中元素都是整数，并邀请读者将这些函数一般化为支持任意类型。表基于数组的实现所使用结构体的声明如下

```
typedef struct {
    int A[MAX];
    int length;
} LIST;
```

这里的LIST是包含两个字段的结构体，第一个字段是存储元素的数组A，而第二个则是含有表中当前元素数目的整数变量length。MAX是个用户定义的常量，用于为存储在表中的元素的数目确定边界。

与表的链表表示相比，基于数组的表示从多个方面讲都更方便。不过，它会受到表不能长

过数组的限制，这可能导致插入操作失败。在链表表示中，只要有可用的计算机内存，就可以让表增长到尽可能长。

对基于数组的表执行词典操作，所花的时间与对链表表示的表执行这些操作所花的时间基本相同。要插入x，先查找x。如果没找到x，就要检查是否有 $length < MAX$。如果$length$不小于MAX，就有出错的情况，因为没法将新元素装入数组中。否则，我们将x存储在`A[length]`中，并将$length$增加1。要删除x，还是先查找x，如果找到，就将数组`A`中x之后的元素都下移一个位置，然后将$length$减1。插入和删除的具体函数实现留作本节习题。接下来要介绍查找操作的细节。

6.5.1 线性查找

图6-11是实现查找操作的函数。因为数组`A`可能很大，所以选择传递指向`LIST`类型结构体的指针`pL`作为*lookup*的形式参数。在该函数中，结构体的两个字段可以称为`pL->A[i]`和`pL->length`。

从 $i = 0$ 开始，第(1)至第(3)行的`for`循环会依次检查数组的每个位置，直到它到达最后出现的位置，或是找到x。如果找到x，就返回`TRUE`。如果它检查了表中的每个元素而没有找到x，就会在第(4)行返回`FALSE`。这种查找方法叫作线性查找或顺序查找。

```
    BOOLEAN lookup(int x, LIST *pL)
    {
        int i;

(1)     for (i = 0; i < pL->length; i++)
(2)         if (x == pL->A[i])
(3)             return TRUE;
(4)     return FALSE;
    }
```

图6-11 通过线性查找进行查找操作函数

不难理解，如果x在表中，那么在找到x之前，平均要查找数组`A[0..length-1]`的一半。因此，设n为`length`的值，那么执行一次查找要花 $O(n)$ 的时间。如果x未出现，就要查找完整个数组`A[0..length-1]`，再次需要 $O(n)$ 的时间。这样的表现，与对用链表表示的表执行查找操作的表现是一样的。

常数因子在实际应用中的重要性

纵观第3章，我们一直在强调运行时间的大O度量的重要性，而且可能给大家留下了这样的印象：大O是唯一的影响因素，或是说任何 $O(n)$ 算法在执行某项任务时都和其他 $O(n)$ 算法有着同样的表现。不过在这里，在对哨兵的讨论中，以及其他几节中，我们都会细究隐藏在 $O(n)$ 之中的常数因子。原因很简单。尽管运行时间的大O度量主导了常数因子，但研究该主题的人都能很快地了解这一点。例如，我们了解到只要n大到足以产生影响，就要使用具有 $O(n\log n)$ 运行时间的排序。软件性能上的竞争优势，往往源于对具有正确“大O”运行时间的算法中的常数因子的改进，而这种优势通常能决定软件产品的成败。

6.5.2　带哨兵的查找

通过将x临时插入表的末尾，可以简化图6-11中for循环的代码，并为该程序提速。在表末端的这个x就叫作哨兵（sentinel）。这项技术最先是在3.6节附注栏内容“更具防御性的程序设计”中提到的，而它在这里有着重要的应用。假设在表的末端始终有一个额外的槽，就可以使用图6-12中的程序查找x。该程序的运行时间仍为$O(n)$，但比例常数更小，因为图6-12所示程序的循环体和循环测试所需的机器指令数，通常小于图6-11所示程序所需的。

```
      BOOLEAN lookup(int x, LIST *pL)
      {
          int i;

(1)       pL->A[pL->length] = x;
(2)       i = 0;
(3)       while (x != pL->A[i])
(4)           i++;
(5)       return (i < pL->length);
      }
```

图6-12　进行带哨兵查找的函数

第(1)行将哨兵放置在刚好越过该表的位置。请注意，因为$length$不会发生改变，所以这个x并非真正是表的一部分。第(3)和第(4)行的循环会增加i，直到我们找到x。请注意，因为设置了哨兵，所以即便表是空表，还是保证能找到x。在找到x之后，第(5)行会测试是找到了表中真正出现的x（也就是，$i < length$），还是找到了哨兵（也就是，$i = length$）。请注意，如果使用哨兵，就一定要严格保证$length$小于MAX，否则就没有位置放置哨兵了。

6.5.3　利用二叉查找对已排序表进行查找

假设表L中的元素a_0、a_1、…、a_{n-1}已经按照非递减次序排好序。如果该已排序表存储在数组A[0..n-1]中，就可以利用二叉查找技术，从而带来可观的速度提升。我们首先必须找到中间元素的下标m，也就是说$m = \lfloor (n-1)/2 \rfloor$。[①]然后将元素$x$与$A[m]$相比较。如果它们相等，就已经找到$x$了。如果$x < A[m]$，就递归地重复对子表A[0..m-1]的查找。如果$x > A[m]$，就递归地重复对子表A[m+1..n-1]的查找。无论何时尝试查找空表，都会报错。图6-13展示了分区过程。

函数binsearch的代码要将x放置在如图6-14所示的已排序数组A中。该函数使用变量low和high表示x可能出现的区域的下界和上界。如果较低的区域超过了较上的区域，那么就没找到x，此时函数就会终止并返回FALSE。

否则，binsearch会通过$mid = \lfloor (low + high)/2 \rfloor$计算该区域的中点。然后该函数会检查区域正中的元素$A[mid]$，以确定$x$是否在该位置。如果$x$不在该位置，而且小于$A[mid]$，就继续在中点下方的区域查找，要是$x$大于$A[mid]$，就继续在中点上方的区域查找。这一思路概括了图6-13所示的划分，其中low是0，而$high$是$n-1$。

---

① $\lfloor a \rfloor$表示a向下取整，就是a的整数部分。因此$\lfloor 6.5 \rfloor = 6$，而且$\lfloor 6 \rfloor = 6$。而$\lceil a \rceil$表示a向上取整，是大于等于a的最小整数。例如$\lceil 6.5 \rceil = 7$，而$\lceil 6 \rceil = 6$。

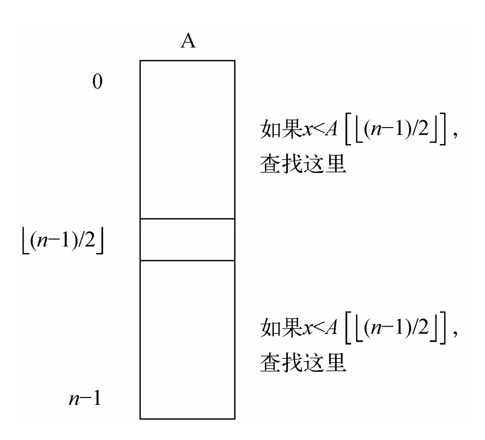


图6-13 二叉查找将区域一分为二

利用归纳断言“如果x在数组中，那么它一定出现在`A[low..high]`这个区域内”，就可以证明函数`binsearch`的正确性。证明过程要对 $high-low$ 的差进行归纳，这留作本节的习题。在每次迭代中，*binsearch*要么

(1) 在到达第(8)行时找到元素x，要么

(2) 在第(5)行或第(7)行，对长度至多为待查找数组`A[low..high]`长度一半的子表递归调用自身。

```
     BOOLEAN binsearch(int x, int A[], int low, int high)
     {
         int mid;

(1)      if (low > high)
(2)          return FALSE;
         else {
(3)          mid = (low + high)/2;
(4)          if (x < A[mid])
(5)              return binsearch(x, A, low, mid-1);
(6)          else if (x > A[mid])
(7)              return binsearch(x, A, mid+1, high);
             else /* x == A[mid] */
(8)              return TRUE;
         }
     }
```

图6-14 使用二叉查找进行查找操作的函数

由长度为n的数组开始，在其长度变为1之前，我们最多对有待查找的数组进行 $\log_2 n$ 次分割。于是我们要么在`A[mid]`找到x，要么在对空表调用该函数后仍未找到x。

想要在具有n个元素的数组`A`中寻找x，可以调用`binsearch(x,A,0,n-1)`。我们知道`binsearch`最多会调用自身 $O(\log n)$ 次。在每次调用中，都要花费 $O(1)$ 的时间，再加上递归调用的时间，因此二叉查找的运行时间是 $O(\log n)$ 。这是可与平均花费 $O(n)$ 时间的线性查找相媲美的查找方式。

6.5.4　习题

(1) 编写函数，利用对数组的线性查找进行下列操作：(a)向表L中插入x；(b)从表L中删除x。

(2) 使用带哨兵的数组，重复习题(1)的练习。

(3) 使用已排序数组，重复习题(1)的练习。

(4) 假设表中元素具有任意类型`ETYPE`，对`ETYPE`类型来说有函数$eq(x, y)$区分x和y是否相等，还有$ly(x, y)$函数可以区分x就`ETYPE`类型元素的次序而言是否先于y，编写以下函数：

　(a) 图6-11中的`lookup`函数；

　(b) 图6-12中的`lookup`函数；

　(c) 图6-14中的`binsearch`函数。

(5) ** 设图6-14中的二叉查找算法最多进行k次探测（也就是在第(3)行求`mid`的值）时最长数组的长度$(high - low + 1)$为$P(k)$。例如，$P(1) = 1$，且$P(2) = 3$。写出$P(k)$的递推关系，再求出自己所写的递推关系的解。它是否说明二叉查找进行的探测次数为$O(\log n)$？

(6) * 对low和$high$之差进行归纳，证明：如果x在区域`A[low..high]`中，那么图6-14中的二叉查找算法会找到x。

(7) 假设数组中可以出现重复的元素，使得插入操作可以在$O(1)$时间内完成。为这种数据结构编写插入、删除和查找函数。

(8) 使用迭代，重新编写二叉查找程序。

(9) ** 为对n个元素的数组进行二叉查找的运行时间建立递推关系，并求解。提示：为了简化问题，可以取$T(n)$作为对具有n个或更少元素（而不是像我们常用的方法那样刚好有n个元素）的数组进行二叉查找的运行时间上界。

(10) 在三分查找中，给定从low到$high$的区域，先计算该区域中大约1/3处的位置

$$first = \lfloor (2 \times low + high) / 3 \rfloor$$

并将其与$A[second]$相比较。如果$x > A[first]$，就计算近似2/3处的位置

$$second = \lceil (low + 2 \times high) / 3 \rceil$$

并将x与$A[second]$进行比较。因此我们将x隔离在这3个区域的其中一个里，每个区域都不大于low到$high$形成区域的三分之一。编写函数执行三分查找。

(11) ** 用三分查找重复习题(5)。也就是说，要找出三分查找时最多需要k次探测的最大数组的递推关系并为其求解。二叉查找和三分查找哪种所需的探测次数多？也就是说，对于给定的k，二叉查找和三分查找哪种能处理更大的数组？

6.6　栈

栈是基于表数据模型的抽象数据类型，栈中的所有操作都是在表的一端执行的，而这一端就叫作栈的栈顶。术语“LIFO表”（后入先出表）指的就是栈。

栈的抽象模型与表的抽象模型如出一辙，也就是一列某一类型的元素a_1、a_2、…、a_n。将栈与一般表区分开来的就是栈可以接受的一些特殊操作。我们将在后面的内容中介绍更加齐全的操作，不过现在，我们注意到最精髓的栈操作就是$push$（压入）和pop（弹出），其中$push(x)$是将元素x放在栈顶，pop则是从栈中移除最顶端的元素。如果将栈顶写在右端，那么对表（$a_1, a_2, \cdots, a_n$）应用$push(x)$操作，就得到表（$a_1, a_2, \cdots, a_n, x$）。而弹出表（$a_1, a_2, \cdots, a_n$）得到的是表（$a_1, a_2, \cdots, a_{n-1}$）。弹出空表$\epsilon$是不可能的，而且会出错。

✦ 示例 6.9

很多编译器首先会把出现在程序中的中缀表达式转换成等价的后缀表达式。例如，表达式 $(3+4)\times 2$ 的后缀形式就是 $34+2\times$。栈可以用来为后缀表达式求值。由空栈开始，我们从左至右扫描需要求值的后缀表达式。每当遇到一个参数，就将其压入栈中。而在遇到运算符时，就弹出栈两次，并记下弹出的操作数。然后对弹出的两个值（其中第二个值是运算符左边的操作数）应用该运算符，然后将结果压入该栈。图6-15展示了处理后缀表达式 $34+2\times$ 每一步操作之后栈的情况。在完成处理后，求值的结果14留在该栈中。

处理的符号	栈	操作
初始化	ϵ	
3	3	*push* 3
4	3,4	*push* 4
+	ϵ	*pop*4；*pop*3
		计算7=3+4
	7	*push* 7
2	7,2	*push* 2
×	ϵ	*pop*2；*pop*7
		计算14=7×2
	14	*push* 14

图6-15 用栈求后缀表达式的值

6.6.1 对栈的操作

之前讨论过的两种抽象数据类型——词典和优先级队列，都拥有一组明确与之关联的操作。栈其实是一些相似的ADT，它们有着相同的底层模型，但各自有着所允许操作集不同的变种。在本节中，我们要讨论栈的通用操作，并展示两种可用来实现栈的数据结构，一种是基于链表的，另一种是基于数组的。

正如之前提到的，在任意一组栈操作中都可以看到*push*和*pop*。为栈ADT选择的操作还有个共性：它们都可以在 $O(1)$ 时间内实现，而与栈中的元素数量无关。大家可以自行验证一下，对于我们提到的两种数据结构，所有操作都只需要常数时间。

除了*push*和*pop*外，通常还需要*clear*操作将栈初始化为空栈。在示例6.9中，默认假设栈一开始为空，而没有解释它为什么是这样。还有一种操作，就是确定栈当前是否为空的测试。

最后要考虑的操作是确定栈是否“已满”的测试。现在在栈的抽象模型中，没有关于满栈的概念，因为原则上讲，栈是可以随意变长的表。不过，在栈的任何一种实现中，都会有某个无法超越的长度。最常见的例子就是在用数组表示表或栈时。正如在6.5节中看到的，必须假设表的长度不会超过常量`MAX`，否则*insert*函数的实现就没法正常工作了。

我们在自己的栈的实现中将要使用的这一操作的正式定义如下。设S是`ETYPE`类型的栈而且x是`ETYPE`类型的元素。

(1) $clear(S)$。将栈S清空。

(2) $isEmpty(S)$。如果S为空，返回`TRUE`，否则返回`FALSE`。

(3) $isFull(S)$。如果S已满，返回TRUE，否则返回FALSE。

(4) $pop(S,x)$。如果S为空，返回FALSE；否则，将x置为栈S栈顶元素的值，并将该元素从栈S中删除，然后返回TRUE。

(5) $push(x,S)$。如果S已满，返回FALSE；否则，将元素x添加到S的栈顶，并返回TRUE。

*pop*的一个常见变种会假设S非空。它只接受S作为参数，并返回被弹出的元素x。而*pop*的另一个版本则根本不返回值，它只是将栈顶处的元素删除。同样，我们可以在编写*push*时假设S“未满”。在这种情况下，*push*不返回任何值。

6.6.2　栈的数组实现

用于表的这种实现也能用于栈。我们将首先讨论基于数组的实现，接着讨论链表表示。在两种情况下，我们都将元素类型定为int。更一般化的工作还是留作本节习题。

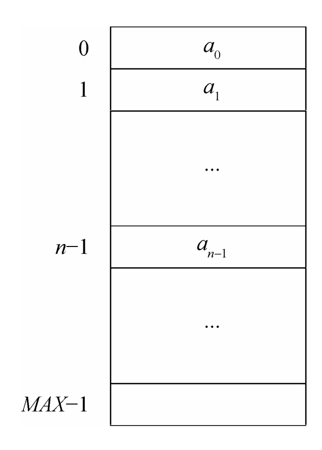


图6-16　表示栈的数组

基于数组的整数栈的声明如下。

```
typedef struct {
    int A[MAX];
    int top;
} STACK;
```

在基于数组的实现中，栈既可以向上增长（从较低区域向较高区域），也可以向下增长（从较高区域向较低区域）。在这里我们选择让栈向上增长[①]，也就是说，栈中最老的元素a_0在位置0，第二老的元素a_1在位置1，而最新插入的元素a_{n-1}在位置$n-1$。

数组结构体中的top字段指示了栈顶的位置。因此，在图6-16中，top的值为$n-1$。空栈是通过top=-1来表示的。在这种情况下，数组A的内容是无关紧要的，栈中没有任何元素。

6.6.1节中定义的5种栈操作对应的程序如图6-17所示。我们通过引用传递栈，来避免复制作为函数参数的大数组。

① 因此“栈顶”在图中是出现在底部的，这是种不凑巧但相当标准的约定。

```
void clear(STACK *pS)
{
    pS->top = -1;
}

BOOLEAN isEmpty(STACK *pS)
{
    return (pS->top < 0);
}

BOOLEAN isFull(STACK *pS)
{
    return (pS->top >= MAX-1);
}

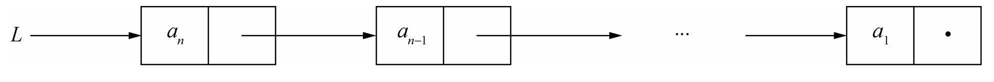
BOOLEAN pop(STACK *pS, int *px)
{
    if (isEmpty(pS))
        return FALSE;
    else {
        (*px) = pS->A[(pS->top)--];
        return TRUE;
    }
}

BOOLEAN push(int x, STACK *pS)
{
    if (isFull(pS))
        return FALSE;
    else {
        pS->A[++(pS->top)] = x;
        return TRUE;
    }
}
```

图6-17　用来实现数组上的栈操作的函数

6.6.3　栈的链表实现

与表一样，可以用链表数据结构表示栈。不过，如果栈顶是表的前端就会很方便。这样的话，可以在表的表头压入和弹出，都只用花 $O(1)$ 的时间。如果必须找到表的端点再压入和弹出，对长度为n的栈执行这些操作就要花 $O(n)$ 的时间。而这样一来，栈S =（$a_1, a_2, \cdots, a_n$）必须用链表"倒着"表示为：



在定义表单元时使用过的类型定义宏也可以用于栈。宏

```
DefCell(int, CELL, STACK);
```

定义了整数栈，并扩展为

```
typdef struct CELL *STACK;
struct CELL {
    int element;
    STACK next;
};
```

对这种表示而言，5种操作可以用图6-18中的函数实现。我们假设`malloc`从不会用尽空间，这

意味着*isFull*操作总是会返回FALSE，而且*push*操作从不会失败。

```
void clear(STACK *pS)
{
    (*pS) = NULL;
}

BOOLEAN isEmpty(STACK *pS)
{
    return ((*pS) == NULL);
}

BOOLEAN isFull(STACK *pS)
{
    return FALSE;
}

BOOLEAN pop(STACK *pS, int *px)
{
    if ((*pS) == NULL)
        return FALSE;
    else {
        (*px) = (*pS)->element;
        (*pS) = (*pS)->next;
        return TRUE;
    }
}

BOOLEAN push(int x, STACK *pS)
{
    STACK newCell;

    newCell = (STACK) malloc(sizeof(struct CELL));
    newCell->element = x;
    newCell->next = (*pS);
    (*pS) = newCell;
    return TRUE;
}
```

图6-18 链表实现的栈所使用的函数

对用链表实现的栈执行*push*和*pop*的效果如图6-19所示。

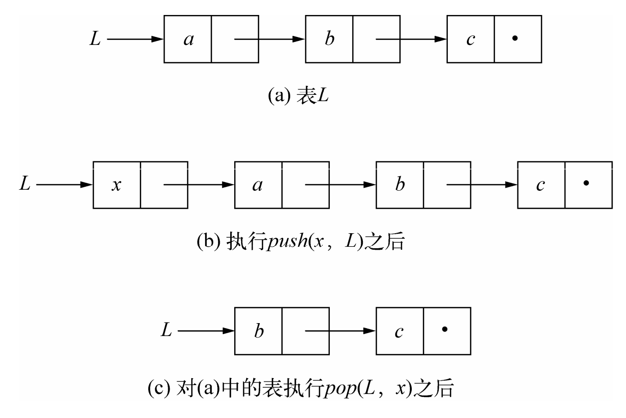


图6-19 对用链表实现的栈执行压入和弹出操作

6.6.4 习题

(1) 由空栈开始，在执行操作序列*push*(*a*)、*push*(*b*)、*pop*、*push*(*c*)、*push*(*d*)、*pop*、*push*(*e*)、*pop*、*pop*之后，栈中还剩什么。

(2) 只使用本节讨论的5种栈操作操作栈，编写C语言程序，按照图6-9所示的算法，为使用整数操作数及4种常用算术运算符的后缀表达式求值。恰当地定义数据类型STACK，并先后在程序中用上图6-17和图6-18中的函数，以此证实大家编写的程序既可以使用数组实现，也可以使用链表实现。

(3) * 怎样用栈为前缀表达式求值?

(4) 计算图6-17和图6-18中各函数的运行时间。它们是否全为$O(1)$？

(5) 栈ADT有时会使用*top*操作，$top(S)$会返回栈S（一定要假设该栈非空）的栈顶元素。编写可与本节中定义栈的

(a) 数组数据结构

(b) 链表数据结构

一起使用的*top*函数。这两个*top*的实现花的时间是否都是$O(1)$？

(6) 模拟栈，计算以下后缀表达式的值：

(a) $ab+cd\times+e\times$

(b) $abcde++++$

(c) $ab+c+d+e+$

(7) * 假设从空栈开始，执行一些压入和弹出操作。如果在这些操作之后的栈为$(a_1, a_2, \cdots, a_n)$，栈顶在右侧，证明：对$i=1, 2, \cdots, n-1$, a_i是在a_{i+1}压入之前被压入栈的。

6.7 使用栈实现函数调用

栈的一项重要应用常不为人所见：栈可以用来为程序中多个函数的变量分配计算机内存空间。我们要讨论的是用于C语言的机制，不过相似的机制也几乎用在其他每种程序设计语言中。

要理解问题是什么，可考虑2.7节中简单的递归阶乘函数fact，该函数图6-20所示。fact函数有一个参数n以及一个返回值。随着fact递归地调用自身，不同的调用将会同时处于活动状态。这些调用有着值各不相同的参数n，而且会产生不同的返回值。那这些有着相同名称的不同对象要存放在哪里呢?

```
        int fact(int n)
        {
(1)         if (n <= 1)
(2)             return 1; /* 依据 */
            else
(3)             return n*fact(n-1); /* 归纳 */
        }
```

图6-20 计算$n!$的递归函数

要回答这一问题，必须先对与程序设计语言相关联的*运行时组织*（run-time organization）有所了解。运行时组织是一种规划，它将计算机内存细分为不同区域，以存放程序所使用的不同数据项。当程序运行的时候，函数的每次执行称作一次*活动*（activation）。与每次活动相关联

的数据对象都存储在计算机内存中称为该活动的*活动记录*（activation record）的区块里。这些数据对象包括参数、返回值、返回地址和该函数的局部变量。

图6-21展示了有代表性的运行时内存细分情况。第一个区域含有执行中的程序的对象代码。而接下来的区域包含了用于该程序的静态数据，比如某些常量以及程序使用的外部变量的值。第三个区域是*运行时栈*，它是向着内存中的高位地址向下增长的。在最高编号内存区域的是*堆*，该区域是为用`malloc`动态分配的对象预留的。[①]

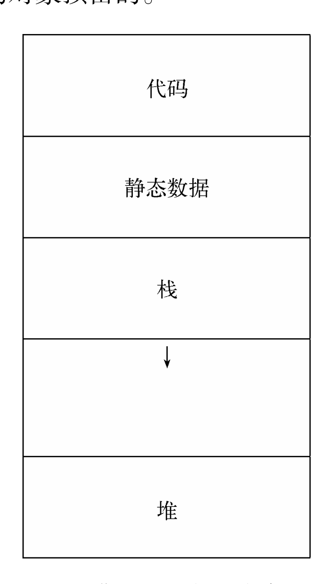


图6-21　典型的运行时内存组织

运行时栈中存放着当前处于活跃状态的所有活动的记录。栈是种合适的结构，因为在调用函数时，可以把活动记录压入栈中。任何时候，当前正在执行的活动A_1的记录会在栈顶位置。而正好位于栈顶之下的是调用A_1的A_2的活动记录。在A_2的活动记录之下的，是调用A_2的活动的记录，以此类推。当函数返回时，就弹出栈顶的活动记录，露出调用该活动的函数的活动记录。这正是要做的事情，因为当函数返回时，控制权会传递给调用函数。

✦　示例 6.10

考虑一下如图6-22所示的程序骨架。该程序是非递归的，而且任一函数中一直只有一个活动。当主函数开始执行时，它包含着变量`x`、`y`和`z`对应空间的活动记录会被压入栈中。当函数`P`在标记为`Here`的位置被调用时，它的活动记录（含有变量`p1`和`p2`对应的空间）会被压入栈中。[②]当`P`调用`Q`时，`Q`的活动记录被压入栈中。至此，栈的情况如图6-23所示。

当`Q`执行完毕时，它的活动记录就会从栈中弹出。此时，`P`也完成了，所以它的活动记录也会被弹出。最后，`main`也完成执行，并将它的活动记录弹出栈。现在栈为空栈，而程序也执行完毕了。

---

① 不要把这里用到的术语“堆”与5.9节中讨论的堆数据结构弄混了。

② 请注意，`P`的活动记录有两个数据对象，因此它的“类型”与主程序活动记录的“类型”是不同的。不过，我们可以将某程序所有记录类型的形式视作某一记录类型的不同变种，因此维护了栈的元素具有相同类型的观点。

```
void P();
void Q();

main() {
    int x, y, z;

    P(); /* 这里 */
}

void P();
{
    int p1, p2;

    Q();
}

void Q()
{
    int q1, q2, q3;
          ...
}
```

图6-22 程序骨架

x y z
p1 p2
q1 q2 q3

图6-23 当函数Q正在执行时的运行时栈

✦ 示例 6.11

考虑图6-20所示的递归函数`fact`。同一时间可能有很多`fact`的活动处于活跃状态，不过每一个活动都有着相同形式的活动记录，即

```
n
fact
```

其中首先装入的是对应参数n的单词，接着是对应返回值的单词，这里表示为`fact`。返回值直到活动的最后一步，在返回之前才会被装入。

假设调用`fact(4)`，这样就创建了具有如下形式的活动记录。

```
n      4
fact   -
```

随着`fact(4)`调用`fact(3)`，接着要将表示该活动的活动记录压入运行时栈，现在该栈就成了：

n	4
fact	-
n	3
fact	-

请注意，这里有两个名为`n`和两个名为`fact`的位置。不过这样并不冲突，因为它们属于不同的活动，而且一次只有一个活动记录可以位于栈顶：属于当前正在执行的活动的活动记录。

`fact(3)`接着会调用`fact(2)`，而`fact(2)`又会调用`fact(1)`。至此，运行时栈如图6-24所示。`fact(1)`现在不再进行递归调用，而是赋值$fact=1$。因此，值1被放入顶部活动记录为`fact`预留的槽中。而其他标记为`fact`的槽未受影响，如图6-25所示。

n	4
fact	-
n	3
fact	-
n	2
fact	-
n	1
fact	-

图6-24　`fact`执行期间的活动记录

n	4
fact	-
n	3
fact	-
n	2
fact	-
n	1
fact	1

图6-25　`fact(1)`计算其值之后

接着，`fact(1)`返回，将对应`fact(2)`的活动记录暴露在外，并在`fact(1)`被调用的位置将控制权返回给`fact(2)`。来自`fact(1)`的返回值1会乘上`fact(2)`对应活动记录中n的值，而该乘积就被放置到该活动记录里`fact`对应的槽中，正如图6-20中第(3)行所需要的。得到的栈如图6-26所示。

n	4
fact	-
n	3
fact	-
n	2
fact	2

图6-26　fact(2)计算其值之后

同样，fact(2)接着将控制权返回给fact(3)，而且对应fact(2)的活动记录会被弹出栈。而返回值2会乘上fact(3)对应的n，得出返回值6。然后，fact(3)返回，并将其返回值乘以fact(4)中的n，得到返回值24。运行时栈现在成了：

n	4
fact	24

至此，fact(4)返回到某假设的调用函数，其活动记录（未表示出来）在栈中正位于fact(4)之下。不过，它会接收返回值24作为fact(4)的值，并继续自己的执行。

习题

(1) 考虑一下图6-27中的C语言程序。main函数的活动记录含有对应整数i的槽。而sum的活动记录中的重要数据包括：

(a) 参数i；

(b) 返回值；

(c) 未命名的临时区域，我们称为temp，用来存储sum(i+1)的值。sum(i+1)是在第(6)行中计算的，而且之后会与$A[i]$相加以形成返回值。

假设$A[i]$的值为$10i$，给出紧邻每次对sum的调用之前和之后活动记录栈的情况。也就是说，给出紧接在压入sum的活动记录之后，且刚要从栈中弹出一个活动记录之前栈的情况。大家无需每次都给出底层（对应main函数的）活动记录的内容。

```
        #define MAX 4
        int A[MAX];
        int sum(int i);

        main()
        {
            int i;

(1)         for (i = 0; i < MAX; i++)
(2)             scanf("%d", &A[i]);
(3)         printf("%d\n", sum(0));
        }

        int sum(int i)
        {
(4)             if (i >= MAX)
(5)                 return 0;
                else
(6)                 return A[i] + sum(i+1);
        }
```

图6-27　习题(1)的程序

(2) *图6-28所示的delete函数会删除链表中第一次出现的整数*x*，链表由定义如下的普通单元组成：

```
DefCell(int, CELL, LIST);
```

delete的活动记录由参数x和pL组成。不过，因为pL是指向表的指针，所以活动记录中第二个参数的值不是指向表中第一个单元的指针，而是另一个指针，它指向的是指向第一个单元的指针。通常，活动记录会存放指向某个单元next字段的指针。在从其他某个函数调用delete(3,&L)，而且*L*是指向链表 (1，2，3，4) 第一个单元的指针时，给出栈的序列。

```
void delete(int x, LIST *pL)
{
    if ((*pL) != NULL)
        if (x == (*pL)->element)
            (*pL) = (*pL)->next;
        else
            delete(x, &((*pL)->next));
}
```

图6-28　习题(2)的程序

6.8　队列

另一种基于表数据模型的抽象数据类型是*队列*。这是一种形式受限的表，它的元素只能从*后端*插入，并从*前端*删除。术语“FIFO表”（先入先出表）就是指队列。

对队列的直观想法就是出纳员窗口前的队伍。人们从尾部进入队伍，并在到达队首时接受服务。与栈不同的是，队列是很公平的，人们是按照进入队伍的顺序接受服务的。因此，等待得最久的那个人就是下一个接受服务的人。

6.8.1　对队列的操作

队列使用的抽象模型与表（或栈）使用的抽象模型是相同的，不过对队列执行的操作却是特殊的。队列具有两种特有的操作，*入队*（enqueue）和*出队*（dequeue）。*enqueue*(*x*)会将*x*添加到队列后端，而*dequeue*则会从队列前端删除元素。就像栈那样，我们还会需要将其他一些实用操作应用到队列上。

设*Q*是元素类型皆为ETYPE的队列，并设*x*是ETYPE类型的元素。我们要考虑以下对队列的操作。

(1) *clear*(*Q*)。将队列*Q*置空。

(2) *dequeue*(Q，*x*)。如果*Q*为空，返回FALSE；否则，将*x*置为*Q*前端元素的值，并将该元素从*Q*中删除，然后返回TRUE。

(3) *enqueue*(*x*，Q)。如果*Q*已满，返回FALSE；否则，将元素*x*添加到*Q*的后端，并返回TRUE。

(4) *isEmpty*(Q)。若*Q*为空则返回TRUE，否则返回FALSE。

(5) *isFull*(Q)。若*Q*已满则返回TRUE，否则返回FALSE。

就像栈那样，我们可以给出更具“信任度”的*enqueue*和*dequeue*，其中*enqueue*不会检查队列是否已满，而*dequeue*不会检查队列是否为空。*enqueue*不再返回值，而*dequeue*则只接受*Q*作为参数，并返回被请出队列的值。

6.8.2 队列的链表实现

用于队列的一种实用数据结构是基于链表的。首先是由宏给出的单元定义

```
DefCell(int, CELL, LIST);
```

一如本章前面的内容，假设队列中的元素都是整数，并请读者自己将我们的函数一般化为处理任意元素类型。

队列的元素将存储到链表的单元中。队列本身是具有两个指针的结构，一个指向前端单元，也就是链表的第一个单元；另一个指向后端单元，也就是链表的最后一个单元。这就是说，有如下定义

```
typedef struct {
    LIST front, rear;
} QUEUE;
```

如果队列为空，front就将是NULL，而rear的值就无关紧要了。

更多的抽象数据类型

可以将栈和队列加到5.9节中引入的抽象数据类型表中。我们在6.6节中介绍了栈使用的两种数据结构，并在6.8节中介绍了队列使用的一种数据结构。而6.8节的习题(3)则提到了实现数组的另一种数据结构，“循环数组”。

抽象数据类型	栈	队　列
抽象实现	表	表
数据结构	1) 链表	1) 链表
	2) 数组	2) 循环数组

图6-29给出了实现本节所提队列操作的程序。请注意，在使用链表时，就没有“满”队列的概念了，这样一来*isFull*总会返回FALSE。不过，如果使用某种基于数组的队列实现，就可能会有满队列。

6.8.3 习题

(1) 给出从空队列开始，在执行操作序列*enqueue*(*a*)、*enqueue*(*b*)、*dequeue*、*enqueue*(*c*)、*enqueue*(*d*)、*dequeue*、*enqueue*(*e*)、*dequeue*、*dequeue*之后剩下的队列。

(2) 证明图6-29中的各函数都能在$O(1)$时间内执行，而不需要考虑队列的长度。

(3) * 可以用数组表示队列，只要队列不会增长得太长。为了让操作只花$O(1)$的时间，必须将数组视为循环的。也就是说，数组A[0..n-1]要被视为A[1]在A[0]之后、A[2]在A[1]之后，以此类推，直到A[n-1]在A[n-2]之后，不过还有A[0]在A[n-1]之后。队列可以用表示队列前端元素和后端元素位置的一对整数front和rear表示。空队列可以表示为在循环的情况下，*front*的位置紧邻*rear*之后，例如$front = 23$且$rear = 22$，或是$front = 0$且$rear = n-1$。请注意，因此该队列不是有n个元素，否则该条件也可以由*rear*紧跟*front*之后来表示。因此，当该队列有$n-1$个元素，而不是有n个元素时，它就已经满了。假设使用循环数组数据结构，为这些队列操作编写函数，不要忘了检查满队列和空队列。

(4) * 证明：如果$(a_1, a_2, \cdots, a_n)$是以a_1为前端的队列，那么对$i = 1, 2, \cdots, n-1$而言，a_i是在a_{i+1}之前入队的。

```
void clear(QUEUE *pQ)
{
    pQ->front = NULL;
}

BOOLEAN isEmpty(QUEUE *pQ)
{
    return (pQ->front == NULL);
}

BOOLEAN isFull(QUEUE *pQ)
{
    return FALSE;
}

BOOLEAN dequeue(QUEUE *pQ, int *px)
{
    if (isEmpty(pQ))
        return FALSE;
    else {
        (*px) = pQ->front->element;
        pQ->front = pQ->front->next;
        return TRUE;
    }
}

BOOLEAN enqueue(int x, QUEUE *pQ)
{
    if (isEmpty(pQ)) {
        pQ->front = (LIST) malloc(sizeof(struct CELL));
        pQ->rear = pQ->front;
    }
    else {
        pQ->rear->next = (LIST) malloc(sizeof(struct CELL));
        pQ->rear = pQ->rear->next;
    }
    pQ->rear->element = x;
    pQ->rear->next = NULL;
    return TRUE;
}
```

图6-29　实现链表队列操作的例程

6.9　最长公共子序列

本节专门探讨一个与表有关的有趣问题。假设有两个表，而我们想知道这两者之间有何差异。该问题会以很多不同的形式出现，也许最常见的就是两个表分别表示某文本文件的两个版本，并希望确定两个版本有哪几行相同的情况。为简便起见，纵贯本节我们都将假设这些表是字符串。

考虑这一问题的一种实用方式就是将两个文件当作符号序列，$x = a_1 \cdots a_m$ 和 $y = b_1 \cdots b_n$，其中a_i表示第一个文件中的第i行，而b_j表示第二个文件的第j行。因此，像a_i这样的抽象符号其实也许是个“大”对象，有可能是一整句话。

UNIX命令`diff`就可以比较两个文本文件的区别。文件x可能是某程序当前的版本，文件y则可能是该程序在经过某次细小修改之前的版本。可以使用`diff`提醒自己在将y变为x时进行的修改。对文本文件的常见修改有：

(1) 插入一行；

(2) 删除一行。

而文本行的修改可以视为删除一行之后紧接着插入一行。

通常，当一个文本文件转化为另一个时，如果对两个文本文件间发生的少量改变加以检验，很容易发现哪些文本行是与哪些行对应的，且很容易看出哪些文本行被删除了以及哪些文本行是新插入的。`diff`命令作出了这样的假设，用两个表表示两个文本文件，表中元素是文本文件中的文本行，于是可以通过首先找出两个表的*最大公共子序列*（Longest Common Subsequence，LCS）来确定都有哪些改变。LCS表示那些没有修改过的文本行。

回想一下，子序列是在保留剩余元素次序的前提下从表中删除0个或多个元素得到的。两个表的*公共子序列*就是同为两者子序列的表，而两个表的最长公共子序列就是两个表公共子序列中最长的那个。

✦ 示例 6.12

在下文的内容中，我们可以将`a`、`b`或`c`这样的字符视为表示文本文件中的文本行，或者如果愿意的话，视作其他类型的元素。举例来说，`baba`及`cbba`都是`abcabba`和`cbabac`的最长公共子序列。可以看到，`baba`是`abcabba`的子序列，因为从`abcabba`中选取位置2、4、5和7就能形成`baba`。字符串`baba`也是`cbabac`的子序列，因为可以选择位置2、3、4和5。同样，`cbba`是由`abcabba`的位置3、5、6和7形成的，也是由`cbabac`的位置1、2、4和5形成的。因此`cbba`也是这些字符串的公共子序列。我们必须相信，这就是最长公共子序列，也就是说，没有长度为5或更长的公共子序列了。这一事实可由接下来要描述的算法得出。

6.9.1 对LCS计算的递归

我们提供了两个表LCS长度的递归定义。该定义使LCS长度的计算变得很容易，而且，通过检验它构建的表，可以发现一个可能的LCS，而不只是其长度。由该LCS，可以推断出文本文件发生了什么变化，本质上讲，不属于LCS的部分都是变化。

要找出表x和表y某个LCS的长度，就需要弄清所有前缀（一个来自x，另一个来自y）对LCS的长度。回想一下，前缀是以表首字母开头的子表，也就是说，`cbabac`的前缀就是ϵ、`c`、`cb`、`cba`，等等。假设$x=(a_1, a_2, \cdots, a_m)$而且$y=(b_1, b_2, \cdots, b_n)$。对每个$i$和每个$j$，其中$i$在0到$m$之间，而$j$在0到$n$之间，都可以要求来自$x$的前缀$(a_1, \cdots, a_i)$和来自$y$的前缀$(b_1, \cdots, b_j)$的LCS。

如果i或j之中有一个为0，那么其中一个前缀就是ϵ，这样两个前缀唯一可能的公共子序列就是ϵ。因此，当i或j之中有一个为0时，LCS的长度就是0。这一直观结果可以转化为在LCS计算方式的非正式讨论之后的归纳中依据和规则(1)的正式形式。

现在考虑一下i和j都大于0的情况。最好将LCS视为两个字符串的某些位置间的匹配。也就是说，对LCS的每个元素而言，都可以将该元素在两个字符串中各自所在的位置匹配起来。匹配过的位置必须具有相同的符号，而且匹配过的位置之间的文本行一定不能交叉。

✦ 示例 6.13

图6-30a展示了字符串abcabba和cbabac两种可能匹配中对应公共子序列baba的那种，图6-30b则展示了对应cbba的匹配。

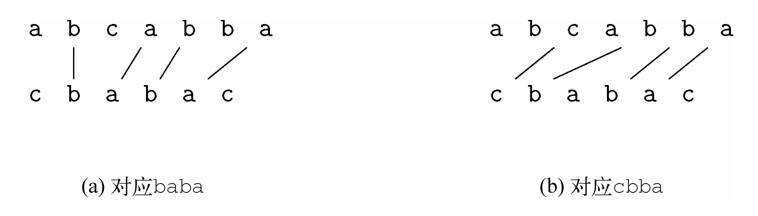


图6-30 表示为位置间匹配的LCS

因此，我们来考虑前缀 $(a_1, \cdots, a_i)$ 和 $(b_1, \cdots, b_j)$ 之间的匹配。存在如下两种情况，具体取决于两个表的最后一个符号是否相等。

(a) 如果 $a_i \neq b_j$，那么匹配中就不可能同时含有a_i和b_j，因此 $(a_1, \cdots, a_i)$ 和 $(b_1, \cdots, b_j)$ 的LCS一定是下列两者之一。

(i) $(a_1, \cdots, a_{i-1})$ 和 $(b_1, \cdots, b_j)$ 的LCS；

(ii) $(a_1, \cdots, a_i)$ 和 $(b_1, \cdots, b_{j-1})$ 的LCS。

如果已经得出了上述两对前缀的LCS的长度，就可以取其中的较大值作为 $(a_1, \cdots, a_i)$ 和 $(b_1, \cdots, b_j)$ 的LCS的长度。这种情况将成为接下来的归纳中正式的规则(2)。

(b) 如果 $a_i = b_j$，就可以匹配a_i和b_j，而且该匹配将不会妨碍其他任何可能的匹配。因此，$(a_1, \cdots, a_i)$ 和 $(b_1, \cdots, b_j)$ 的LCS的长度，要比 $(a_1, \cdots, a_{i-1})$ 和 $(b_1, \cdots, b_{j-1})$ 的LCS的长度大1。这种情形将成为接下来的归纳中正式的规则(3)。

这些直观结果让我们给出了$L(i, j)$——$(a_1, \cdots, a_i)$ 和 $(b_1, \cdots, b_j)$ 的LCS的长度——的递归定义。其中利用了对$i+j$的和的完全归纳。

依据。如果$i+j = 0$，那么i和j都为0，所以LCS是ϵ。因此$L(0,0) = 0$。

归纳。考虑i和j，并假设已经为满足 $g+h<i+j$ 的任意g和h计算出$L(g, h)$。有如下3种情况需要考虑。

(1) 如果i或j中有一个为0，那么$L(i, j) = 0$。

(2) 如果 $i>0$ 且 $j>0$，而且 $a_i \neq b_j$，那么$L(i, j) = \max\big(L(i, j-1), L(i-1, j)\big)$。

(3) 如果 $i>0$ 且 $j>0$，而且 $a_i = b_j$，那么$L(i, j) = 1+L(i-1, j-1)$。

6.9.2 用于LCS的动态规划算法

我们最终想要的是$L(m, n)$，即表x和表y的LCS的长度。如果根据之前的归纳编写递归程序，它所花的时间是m和n的较小者的指数。这一简单的递归算法对 $n = m = 100$ 这样的情况而言要花上太多太多的时间。这一递归表现如此糟糕的原因有些复杂。首先，假设表x和表y中的字符间完全没有匹配，并调用$L(3,3)$。这会带来对$L(2,3)$和$L(3,2)$的调用，而这两次调用又都会带来对$L(2,2)$的调用。因此就要将$L(2,2)$的工作完成两遍。随着L的参数变小，$L(i,j)$的调用次数会迅速增加。如果将调用追踪继续下去，就会发现，$L(1,1)$被调用了6次，$L(0,1)$和$L(1,0)$各被调用了10次，而$L(0,0)$被调用了20次。

如果构建二维表或二维数组来存储对应不同i和j的$L(i, j)$，就可以有更佳的表现。如果按照归纳的次序计算这些值，也就是说先从最小的$(i+j)$的值开始计算，那么在计算$L(i, j)$时所需的L的值总是在表中。其实，逐行计算L要更简单，也就是对i =0，1，2等计算L，而在一行之中，要按列计算L，也就是对$j = 0$，1，2等计算L。在计算$L(i,j)$时，还是一定能够在表中找到所需的值，而且不需要进行递归调用。这样一来，计算表中每一项都只需要花费 $O(1)$ 的时间，而要构建二维表表示长度分别为m和n的表的LCS，需要花的时间是 $O(mn)$ 。

图6-31展示了填充该表的C语言代码，是按行处理而非按$i+j$ 的和处理的。假设表x存储在数组$A[1..m]$中，而表y存储在$b[1..n]$中。请注意，数组中标号为0的元素未被使用，这样做简化了图6-31中的表示法。这里将证明该程序处理长度分别为m和n的表的运行时间为 $O(mn)$ 的工作留作本节习题。[①]

```
for (j = 0; j <= n; j++)
    L[0][j] = 0;
for (i = 1; i <= m; i++) {
    L[i][0] = 0;
    for (j = 1; j <= n; j++)
        if (a[i] != b[j])
            if (L[i-1][j] >= L[i][j-1])
                L[i][j] = L[i-1][j];
            else
                L[i][j] = L[i][j-1];
        else /* a[i] == b[j] */
            L[i][j] = 1 + L[i-1][j-1];
}
```

图6-31　填充LCS表的C语言程序片段

动态规划

术语“动态规划”源自R. E. Bellman在1950年为解决控制系统中的问题所提出的一般理论。而人工智能领域的工作者通常会将这一技术称为备忘（memoing）或制表（tabulation）。

像本例这样的填表技术通常称为*动态规划算法*（dynamic programming algorithm）。这种情况下，它比重复为相同子问题求解的直接递归实现更高效。

✦　示例 6.14

设x是表`cbabac`，y是表`abcabba`。图6-32展示了为这两个表构建的二维表。例如，$L(6,7)$是 $a_6 \neq b_7$ 的情况。因此$L(6,7)$就是它下方和左侧两个项中的较大者。因为这两项分别为4和3，所以我们右上角的项$L(6,7)$置为4。现在考虑一下$L(4,5)$。因为a_4和a_5都是符号`b`，所以在$L(4,5)$左下的$L(3,4)$这项上加1。因为该项为2，所以将$L(4,5)$置为3。

---

① 严格地讲，我们只讨论了是一个单变量函数的大O表达式。不过，这里要表达的意思应该是明了的。如果$T(m,n)$是该程序处理长度为m和n的表的运行时间，那么存在常数m_0、n_0和c，使得对所有的 $m \geqslant m_0$ 和 $n \geqslant n_0$ ，都有 $T(m,n) \leqslant cmn$ 。

c	6	0	1	2	3	3	3	3	4
a	5	0	1	2	2	3	3	3	4
b	4	0	1	2	2	2	3	3	3
a	3	0	1	1	1	2	2	2	3
b	2	0	0	1	1	1	2	2	2
c	1	0	0	0	1	1	1	1	1
	0	0	0	0	0	0	0	0	0
		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a

图6-32　对应cbabac和abcabba最长公共子序列的二维表

6.9.3　LCS的恢复

现在就得到了能给出LCS长度的二维表，不仅能给出问题中两个表的LCS的长度，而且可以给出它们每对前缀的LCS的长度。从这些信息一定能推导出问题中两个表可能的LCS之一。要完成这一工作，就要找到形成LCS之一的匹配元素对。我们会找到一条从右上角开始穿越该二维表的路径，而这一路径将确定一个LCS。

假设这条从右上角开始的路径已经将我们带到了第i行第j列，也就是该二维表中对应元素对a_i和b_j的点。如果$a_i = b_j$，$L(i, j)$就是$1 + L(i-1, j-1)$。因此可以将a_i和b_j当作已匹配的元素对，而且我们会把a_i（也是b_j）表示的符号包含在LCS中，并排在目前为止所有已被找到的LCS元素之前。然后将路径向左下移动，也就是说移动到第i–1行第j–1列。

不过，也可能$a_i \neq b_j$。如果这样，$L(i, j)$肯定至少与$L(i–1, j)$和$L(i, j–1)$中的某一个是相等的。如果$L(i, j) = L(i–1, j)$，就会把路径向下移动一行，否则，就知道$L(i, j) = L(i, j–1)$，就会把路径向左移动一列。

遵循这一规则，最终会到达左下角。至此，就已经选定了一个作为LCS的元素序列，而且该LCS本身也是由这些元素组成的表，表中元素的次序与它们被选定的次序是相反的。

c	6	0	1	2	3	3	3	3	**4**
a	5	0	1	2	2	3	3	3	**4**
b	4	0	1	2	2	2	3	**3**	3
a	3	0	1	1	1	2	**2**	2	3
b	2	0	0	1	1	1	**2**	2	2
c	1	0	0	0	**1**	**1**	1	1	1
	0	**0**	**0**	**0**	0	0	0	0	0
		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a

图6-33　找到最长公共子序列cbba的路径

✦　示例 6.15

图6-33再次展示了图6-32中的二维表，并将路径加粗表示出来。我们从值为4的$L(6,7)$开始。因为$a_6 \neq b_7$，所以立刻向左和向下寻找值4，它至少会在这两个位置中的一个出现。在本例中，

4出现在$L(6,7)$下方，所以我们移动到$L(5,7)$。现在有 $a_5 \neq b_7$ ，都是a。因此a就是该最长公共子序列中最后的符号，于是我们移向左下方，移动到$L(4,6)$。

因为a_4和b_6都是b，所以将b放入正在成形的LCS中，位于a之前，而我们继续向左下移动，移动至$L(3,5)$。这里，我们发现 $a_3 \neq b_5$ ，不过$L(3,5)$的值为2，与它下面和左边的项都相等。在这种情况下我们选择向下移动，所以接下来要移动到$L(2,5)$。在该点会看到 $a_2 = b_5 = \text{b}$，所以我们将b放在正在成形的LCS前头，并继续向左下移动到$L(1,4)$。

因为 $a_1 \neq b_4$ ，$L(1,4)$只有在它左边的项，而且该项有着与它相同的值1，所以我们移动到$L(1,3)$。现在有 $a_1 = b_3 = \text{c}$，因此可以将c添加到LCS的前端，并移动到$L(0,2)$。至此，我们别无选择，只能向左移动到$L(0,1)$，然后移动到$L(0,0)$，最终完成了这条路径。得到的LCS由我们发现的4个字符按反序组成，就是cbba。这刚好是我们在示例6.12中提到的两个LCS之一。要得到其他的LCS，可以在$L(i,j)$与$L(i-1,j)$和$L(i,j-1)$都相等时选择向左移动而非向右移动，并且在$L(i-1,j)$和$L(i, j-1)$其中之一等于$L(i,j)$时，选择向左或向右移动，即便是在 $a_i = b_j$ 的情况下（即跳过某些匹配而直接到达其左边的匹配）。

可以证明，这一寻路算法总能找到最大公共子序列。我们要利用对两个表长度之和进行完全归纳加以证明的命题如下。

命题 $S(k)$。如果在第i行第j列，其中 $i + j = k$ ，而且有 $L(i, j) = v$ ，我们随后就会在LCS中找到v个元素。

依据。依据是 $k = 0$ 的情况。如果 $i + j = 0$ ，那么i和j都为0。我们已经完成了路径，并发现LCS不会有更多元素。因为已经知道 $L(0,0) = 0$ ，所以归纳假设对 $i + j = 0$ 成立。

归纳。假定对k或更小的和的归纳假设成立，并令 $i + j = k + 1$ 。假设我们在值为v的 $L(i, j)$ 处。如果 $a_i = b_j$ ，就找到了匹配并移动到 $L(i-1, j-1)$ 。因为 $(i-1)+(j-1)$ 的和小于 $i+1$ ，所以归纳假设是适用的。因为 $L(i-1, j-1)$ 一定是$v-1$，所以我们知道LCS还将找到$v-1$个元素，再加上已经找到的一个元素，就会给我们v个元素。这一直观结果证明了这种情况下的归纳假设。

唯一的例外是 $a_i \neq b_j$ 的情况。这种情况下，$L(i-1, j)$ 或 $L(i, j-1)$ ，或者这两者，一定具有值v，而且我们要移动到具有值v的这些位置之一。因为任一情况下行列值的和都是 $i + j - 1$ ，所以归纳假设是适用的，这样就能得出在LCS中找到v个元素的结论。这样我们又能得出 $S(k+1)$ 为真的结论。因为已经考虑了所有的情况，所以就完成了证明，并可以说如果在数据项 $L(i, j)$ 处，就总是在LCS找出 $L(i, j)$ 个元素。

6.9.4 习题

(1) 下列表的LCS的长度各为多少？

(a) banana和cabana

(b) abaacbacab和bacabbcaba

(2) * 找到习题(1)两个小题中两个表的所有LCS。提示：在为习题(1)构建二维表之后，从右上角往回追溯，在遇到有两条或三条不同路径的点时，要顺着每种选择继续移动。

(3) ** 假设使用我们最先描述的递归算法而不是推荐的填表程序计算LCS。如果对两个没有共同符号的表调用$L(4,4)$，要执行多少次对$L(1,1)$的调用？提示：使用填表（动态规划）算法计算二维表，给出对应所有i和j的 $L(i, j)$ 的值。将计算结果与4.5节中的帕斯卡三角相比较。这一关系表示了与调用次数的公式有关的哪些信息？

(4) ** 假设有表x和表y，且二者的长度均为n。当n小到一定程度之后，就最多只有一个字符串是x和y

的LCS了，虽然该字符串可能出现在x和/或y的不同位置。例如，如果$n=1$，那么LCS只能是ϵ，除非x和y都是同一个符号a，这种情况下a就是唯一的LCS。那么，让x和y可以有两个不同LCS的最小n值是多少？

(5) 证明图6-31所示的程序运行时间为$O(mn)$。

(6) 编写C语言程序，接受图6-31所示程序计算出的那种表，并找出LCS在各字符串中的位置。如果该表的规格为$m \times n$，那么这一程序的运行时间是多少？

(7) 在6.9节开头，我们表示过LCS的长度是与两个字符串最大位置匹配的大小有关的。

(a*) 通过对k的归纳证明，如果两个字符串有着长度为k的公共子序列，那么它们也有着长度为k的匹配。

(b) 证明：如果两个字符串有长度为k的匹配，那么它们也有长度为k的公共子序列。

(c) 由(a)和(b)得出，LCS的长度与匹配的最大值其实是一回事。

6.10　字符串的表示

字符串可能是实践过程中最常见的表的形式。表示字符串的方法数不胜数，而且其中一些技巧很难适用于其他类型的表。因此，本节专门介绍一些与字符串有关的特殊问题。

首先，应该意识到存储单个字符串基本不成问题。通常，我们有大量很短的字符串。它们可能形成词典，意味着我们可以随着时间的推移插入和删除字符串，也可能是*静态*字符串集合，时间再久也不会改变。下面要讲两个典型的例子。

(1) *字母索引*是一种研究文本的实用工具，它是由文档中使用过所有单词以及这些单词出现的位置构成的表。在大型文档中通常会有成千上万个不同的单词，而单词每出现一次就要被存储一次。这一单词集合是静态的，也就是说，一旦成形就不会改变，除非原有的字母索引中存在错误。

(2) 将C语言程序转化为机器代码的编译器必须记录表示程序变量的所有字符串。大型程序可能拥有成百上千的变量名。想想看，分别在两个函数中声明的局部变量`i`，其实是两个不同的变量，这样就能明白为什么会有如此多的变量了。随着编译器对程序加以扫描，会找到新的变量名，并将其插入变量名集合中。一旦编译器完成了函数的编译，该函数的变量对随后的函数来说便不可用了，因此可以删除掉。

在这两个例子中，都存在很多短字符串。英语中的短单词比比皆是，而程序员则喜欢用`i`或`x`这样的字母表示变量。另一方面，不管是在英语文本还是在程序中，单词的长度都是没有限制的。

6.10.1　C语言中的字符串

C语言程序中可能出现字符串常量，而它们会被存储为字符数组，后面跟上名为*空字符*（null character）且值为0的特殊字符“`\0`”。不过，在上面提到的应用中，我们需要随着程序运行而创建并存储新字符串的便利。因此，需要能向其中存储任意字符串的数据结构。其中一些可能如下。

(1) 使用定长数组存放字符串。比数组短的字符串之后由空字符补齐。而比数组长的字符串不能完整地存储到数组中，它们必须被*截断*，只将长度与数组长度相等的前缀存储到数组中。

(2) 与(1)类似的模式，但假设每一个字符串或被截断字符串的前缀之后都有一个空字符。这种方式简化了字符串的读操作，但它让数组中可以存储的字符串的长度减少了1。

(3) 与(1)类似的模式，它不会在字符串后放置空字符，而是用另一个整数*length*指示字符串的真实长度。

(4) 要避免最大字符串长度的限制，可以将字符串中的字符存储为链表元素，而且可以将多个字符存储在一个单元中。

(5) 可以创建大型字符数组，将很多单独的字符串放置其中。而每个字符串就由指向该字符串开头字符在该数组中位置的指针表示。字符串可能以空字符结尾，也可能有与之关联的长度。

6.10.2 定长数组表示

我们来考虑一下上述第(1)类结构，其中字符串是由定长数组表示的。在下面的例子中，我们会创建拥有定长数组作为其中一个字段的结构体。

✦ 示例 6.16

考虑一下用来存放索引中某一项，即单个单词以及与其相关的信息的数据结构。我们需要存放下列内容。

(1) 单词本身；

(2) 单词出现的次数；

(3) 表示文档中文本行的表，该单词会在其中出现一次或多次。

因此可以使用如下结构体：

```
typedef struct {
    char word[MAX];
    int occurrences;
    LIST lines;
} WORDCELL;
```

这里的MAX是指单词的最大长度。所有的WORDCELL结构体都包含一个名为word的有MAX个字节的数组，不管要存放的单词到底有多短。

字段occurrence是计算某单词出现次数的计数器，而lines则是指向链表开头的指针。链表中的单元具有由以下宏定义的常规类型：

```
DefCell(int, CELL, LIST);
```

每个单元存放着一个整数，表示出现问题中单词的文本行。请注意，如果某个单词在一行中出现若干次，那么occurrence就要比表的长度更大。

在图6-34中，我们看到表示《圣经・创世记》第1章中的单词earth的结构体。假设MAX至少为6。表示行（诗句）号的完整表是(1，2，10，11，12，15，17，20，22，24，25，26，28，29，30)。

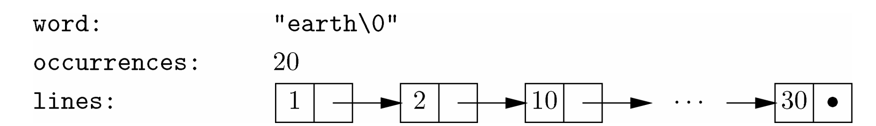


图6-34 单词earth在《圣经・创世记》第1章中的索引项

整个索引可能是由一系列WORDCELL类型的结构体组成的。例如，这些结构体可以被组织为一棵二叉查找树，有着基于单词字母顺序的 < 次序。在使用字母索引时，该结构体可以提供相当高的单词访问速度。而随着我们不断扫描文本，找到并列出各单词的出现，它还能让我们

高效地创建索引。要使用二叉树结构，就需要在类型WORDCELL中定义表示左子节点和右子节点的字段。我们还可以在原始的WORDCELL类型定义中加入“next”字段，从而将这些结构体排列在链表中。这是一种更简单的结构，不过如果单词数量众多，它的效率就要差不少。我们在第7章中将会看到如何在散列表中排列这些结构体，这基本上是解决这一问题的所有数据结构中性能最佳的。

6.10.3　字符串的链表表示

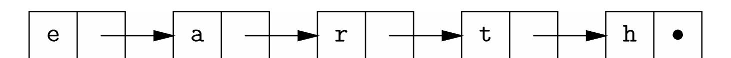
字符串长度的限制，以及不管字符串有多短都需要分配固定量的空间，这是字符串定长数组实现的两大缺点。不过，C语言和其他语言都允许使用者构建其他更为灵活的数据结构来表示字符串。例如，如果希望字符串长度没有上限，可以使用常规的字符链表存放字符串。也就是，可以声明如下类型：

```
typedef struct CHARCELL *CHARSTRING;
struct CHARCELL {
    char character;
    CHARSTRING next;
};
```

在类型WORDCELL中，CHARSTRING成了word字段的类型，如：

```
typedef {
    CHARSTRING word;
    int occurrences;
    LIST lines;
} WORDCELL;
```

例如，单词earth可以表示为

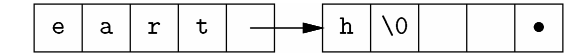


这种模式消除了单词长度的上限，不过在实际应用中，对空间的利用却不是很好。原因在于，假设用1个字节表示字符，并且通常用4个字节表示指向链表下一个单元的指针，那么每个CHARCELL类型的结构体至少要占用5个字节。因此，绝大部分空间是由指针的“系统开销”占用的，只有少部分空间是由字符的“有效负载”使用的。

不过可以更灵活些，将若干字节打包装入每个单元的数据字段。例如，如果在每个单元中放入4个字符，而指针还是消耗4个字节，那么就有一半空间是由“有效负载”使用的，而每单元一个字符的模式只有20%的有效负载。唯一要注意的地方是，必须用某一字符（比如说空字符）作为字符串终止字符，就像存储在数组中的字符串所做的那样。一般而言，如果CPC（Characters Per Cell，每单元字符数）是我们希望在一个单元中放置的字符数，就可以按照如下声明定义单元：

```
typedef struct CHARCELL *CHARSTRING;
struct CHARCELL {
    char characters[CPC];
    CHARSTRING next;
};
```

例如，如果CPC= 4，就可以将单词earth存储在两个单元中，形如：



也可以将CPC增加到4以上。这样做的话，指针所占空间的比例进一步下降，这是很好的情况，意味着使用链表的系统开销下降了。另一方面，如果使用非常大的CPC值，就会发现，几乎所有单词都只需要使用一个单元，但是该单元中可能有很多未使用的位置，就像长度为CPC的数组那样。

✦ 示例 6.17

假设在所有的字符串中，有30%是长为1到4个字符的字符串，40%是长5到8个字符的，20%是9到12个字符的，还有10%是13到16个字符的。图6-35中的表给出了表示4个范围的单词的链表，在CPC分别为4、8、12和16时所占的字节数。就我们假设的单词出现频率而言，CPC= 8的结果最佳，平均要使用15.6个字节。也就是说，每个单元最好用8个字节存放字符，加上存放next指针的4个字节，即每个单元总共要使用12个字节。请注意总空间开销，在加上指向表前端的指针之后，就达到了19.6字节，就不如使用16字节的字符数组那么好了。不过，这种链表模式也可以容纳长度超过16个字符的字符串，虽然这里假设找到这样这种字符串的概率为0。

		每单元字符数			
范围	概率	4	8	12	16
1-4	0.3	8	12	16	20
5-8	0.4	16	12	16	20
9-12	0.2	24	24	16	20
13-16	0.1	32	24	32	20
平均		16.8	15.6	17.6	20.0

图6-35　当CPC为不同值时，不同长度范围的字符串使用的字节数

6.10.4　字符串的海量存储

还存在另一种存储大量字符串的方法，它兼具数组存储的优势（低系统开销）与链表存储的优势（因为填充而不浪费空间，且字符串长度无限制）。我们创建一个非常长的字符数组，并将每个字符串都存储到这一数组中。为了区分一个字符串的结束与下一个字符串的开始，需要一个名为*端记号*（endmarker）的特殊字符。端记号字符不是合法字符串的一部分。尽管选择不打印的字符（比如空字符）是更常见的，不过为了便于识别，在接下来的内容中会使用*作为端记号。

✦ 示例 6.18

假设通过

```
char space[MAX];
```

声明数组space。然后就可以通过给出指向某单词在space数组中第一个位置的指针来存储该单词了。模仿示例6.16中WORDCELL结构体的WORDCELL结构就是：

```
typedef struct {
    char *word;
    int occurrences;
    LIST lines;
} WORDCELL;
```

在图6-36中，我们看到表示《圣经·创世记》的字母索引中单词the的WORDCELL结构体。不过

接下来的情况并不是这样。即便接下来的元素中含有beginning、God和created，space数组也不会出现第二个the了。通过增加WORDCELL结构体中对应单词the的occurrences的值，该单词就被记录在列了。随着在这本书中继续向前处理，发现单词更多的重复，space数组中的项就不再像《圣经》原文那样了。

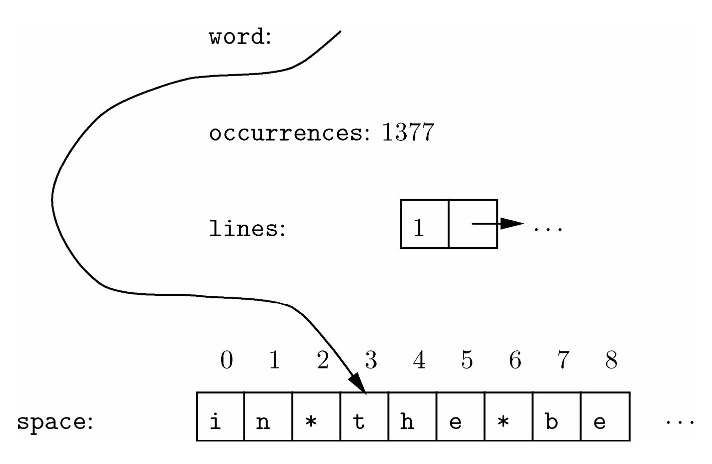


图6-36　通过字符串空间的索引表示单词

就像示例6.16中那样，示例6.18中的结构体也可以通过添加指向WORDCELL结构体的合适指针字段形成二叉查找树或链表这样的数据结构。函数$lt(W_1,W_2)$可以按照两个WORDCELL类型结构体W_1和W_2的word字段的词典顺序对二者加以比较。

要使用这样的二叉查找树构建索引，需要使用指针available指向space数组中第一个未被占用的位置。一开始，available指向space[0]。假设对要构建索引的文本进行扫描，并且找到下一个单词，比方说是the。我们现在不知道the是否已经在二叉查找树中，因此要临时将the*添加到available指向的位置以及接下来的3个位置中。记住，这一新添加的单词要占用4个字节。

现在可以在二叉查找树中查找单词the了。如果找到该单词，就在其出现次数的计数器上加1，并将当前行插入表示文本行的表中。如果未找到，就创建含有WORDCELL结构体的各个字段以及左子节点和右子节点指针（都为NULL）的新节点，并将其插入树中合适的位置。我们将新节点的word字段置为available，这样一来它就指向我们这里单词the的副本了。再将occurrences置为1，并创建由当前文本行的组成的表示字段lines的表。最后，必须给available加上4，因为现在已经把单词the永久地加入space数组了。

空间用尽时会发生什么情况？

我们假设了space是足够大的，从而总是有空间容纳新添加的单词。实际情况是，每当添加新的字符时，我们都必须注意当前写入字符的位置一定要小于MAX。

如果想在空间用尽后输入新的单词，就需要准备好在旧块用尽后获得新空间块。并不是创建数组space，而是要定义字符数组类型

```
typedef char SPACE[MAX];
```

接着可以按照如下定义创建新数组，其中available指向数组的第一个字符。

```
available = (char *) malloc(sizeof(SPACE));
```

只要直接赋值

```
last = available + MAX;
```

就能得到该数组的末端了。

然后可以向available指向的数组插入单词。如果没办法再往该数组中装入单词，就可以调用malloc创建另一个字符数组。当然，一定要注意不要让写操作越过数组的末端，而且如果遇到长度大于MAX的字符串，就没办法以这种模式存储单词了。

6.10.5　习题

(1) 针对示例6.16中讨论的结构体类型WORDCELL，编写如下程序。

(a) 函数create，要返回指向WORDCELL类型结构体的指针。

(b) 函数insert(WORDCELL *pWC, int line)，接受指向WORDCELL结构体的指针以及行号，在该单词的出现次数上加1，而且如果该行未出现在表示各行的表中，就将其添加进去。

(2) 重做示例6.17，假设长度从1到40的单词都等可能出现，也就是10%的单词长度为1至4，10%的为5至8，等等，直到10%的在37到40这个范围内。如果CPC分别为4、8、…、40，分别平均需要多少个字节？

(3) * 在示例6.17的模型中，如果从1到n的所有单词长度都等可能出现，那么CPC为何值（表示为n的函数）时使用的字节数是最少的？如果得不出具体的答案，也可以用大O近似值来表示。

(4) * 使用示例6.18中所示结构体的优势之一在于，在两个或多个单词中可以共享space数组的一些部分。例如，在图6-36所示的数组中，有单词he的word字段等于5。对单词all、call、man、mania、maniac、recall、two、woman进行压缩，使其在space数组中占用尽可能少的元素。通过压缩可以节省多少空间？

(5) * 另一种存储单词的方法要从space数组中消除端记号字符。并且要为示例6.18中的WORDCELL结构体加入length字段，从而表示出在word字段表示的单词中从第一个字符起共有多少个字符。假设length字段的整数要占用4字节，那么这种模式与示例6.18中的模式相比，是节省了空间还是更耗费空间？如果存储该整数只需要1字节呢？

(6) ** 习题(5)中描述的方案也带来了压缩space数组的可能。现在即便单词之间没有任意一方是另一方的后缀，也可以互相重叠了。使用习题(5)中的模式，存储习题(4)表中的单词，需要space数组中多少个单元？

(7) 编写程序，接受示例6.18中讨论的两个WORDCELL结构体，并确定哪个结构体中的单词在词典次序上先于另一个。回想一下，示例6.18中单词都是由*终结的。

6.11　小结

本章涵盖了以下要点。

- 表是一种表示元素序列的重要数据模型。
- 链表和数组是两种可用于实现表的数据结构。
- 表是词典抽象数据类型的一种简单实现，不过其效率无法与第5章中的二叉查找树和第7章中的散列表相提并论。
- 将“哨兵”放置在数组末尾，从而确保找到正在寻找的元素，是一种实用的提高效率的方法。

- 栈和队列都是特殊类型的表。
- 栈是实现递归函数的“幕后英雄”。
- 字符串是表的一种重要特例，而且有若干种特殊的数据结构可以有效地表示字符串，其中包括每单元存储若干字符的链表，以及由很多字符串共享的大型数组。
- 找出最大公共子序列的问题可以通过“动态规划”技术有效地解决，在动态规划过程中我们会按照合适的次序填充信息表格。

6.12 参考文献

Knuth [1968]仍然是表数据结构的基本来源。尽管很难追溯“表”或“栈”这种非常基础的概念的起源，但最先使用表作为数据模型的程序设计语言是IPL-V（Newell etal. [1961]），不过早期的表处理语言中，只有Lisp（McCarthy etal. [1962]）现在还属于重要语言的范畴。顺便提一句，Lisp表示的是LISt Processing（表处理）。

Aho, Sethi, and Ullman [1986]中详细地讨论了栈在递归程序的运行时实现中的使用。

6.9节描述的最长公共子序列算法源自Wagner and Fischer [1975]。Hunt and Szymanski [1977]则描述了实际应用在UNIX的`diff`命令中的算法。Aho [1990]全面介绍了若干种涉及字符串匹配的算法。

Bellman [1957]描述了作为抽象技巧的动态规划。Aho, Hopcroft, and Ullman [1983]给出了一系列使用动态规划的算法的示例。

Aho, A. V. [1990]. “Algorithms for finding patterns in strings,” in *Handbook of Theoretical Computer Science* Vol. A: *Algorithms and Complexity* (J. Van Leeuwen, ed.), MIT Press, Cambridge, Mass.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1983]. *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass.

Aho, A. V., R. Sethi, and J. D. Ullman [1986]. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass.

Bellman, R. E. [1957]. *Dynamic Programming*, Princeton University Press, Princeton, NJ.

Hunt, J. W. and T. G. Szymanski [1977]. “A fast algorithm for computing longest common subsequences,” *Comm. ACM* **20**:5, pp. 350–353.

Knuth, D. E. [1968]. *The Art of Computer Programming*, Vol. I, *Fundamental Algorithms*, Addison-Wesley, Reading, Mass.

McCarthy, J. et al. [1962]. *LISP 1.5 Programmer’s Manual*, MIT Computation Center and Research Laboratory of Electronics, Cambridge, Mass.

Newell, A., F. M. Tonge, E. A. Feigenbaum, B. F. Green, and G. H. Mealy [1961]. *Information Processing Language-V Manual*, Prentice-Hall, Englewood Cliffs, New Jersey.

Wagner, R. A. and M. J. Fischer [1975]. “The string to string correction problem,” *J. ACM* **21**:1, pp. 168–173.

第 7 章

集合数据模型

集合（简称为“集”）是最为基础的数学数据模型。数学中的每种概念，从树到实数，都可以表示为一类特殊的集合。在本书中，我们已经见识过以概率空间中事件的形式出现的集。词典抽象数据类型就是一种集合，可以对其执行插入、删除和查找这些特殊操作。因此，说集合也是计算机科学中的基础模型应该不会让人惊讶。在本章中，我们要了解与集合有关的基本定义，并考虑有效实现集合操作的算法。

7.1　本章主要内容

本章将涵盖以下主题。

- 集合论的基本定义以及集合的基本运算（7.2节和7.3节）。
- 3种最常用于实现集合的数据结构：链表、特征向量和散列表。我们将比较这些数据结构在支持各种集合运算时的效率（7.4节 ~ 7.6节）。
- 作为有序对集合的关系和函数（7.7节）。
- 表示关系和函数的数据结构（7.8节和7.9节）。
- 特殊类型的二元关系，如偏序关系和等价关系（7.10节）。
- 无限集（7.11节）。

7.2　基本定义

在数学中，术语“集合”是没有明确定义的。就像几何中的“点”和“线”那样，集合也是由其属性定义的。具体地说，有只适用于集合的成员概念。当S为集合，而x为任意事物时，我们可以提出如下问题：“x是否为集合S的成员？”集合S就是由所有属于S的成员的元素x组成的。以下几点总结了与集合有关的一些重要概念。

(1) 表达式$x \in S$意味着元素x是集合S的成员。

(2) 如果$x_1, x_1, \cdots, x_n$都是集合S的成员，就可以写为

$$S = \{x_1, x_2, \cdots, x_n\}$$

在这里，每个x都是不同的，在集合中任一元素都是不能重复出现的。然而，集合中各成员的顺序是无关紧要的。

(3) 空集记为$\varnothing$，表示没有任何成员的集合。也就是说，不管x是什么，$x \in \varnothing$都为假。

✦ **示例 7.1**

设$S=\{1,3,\cdots,6\}$，也就是说，S是只含有整数成员1、3、6的集合。我们可以说$1\in S$，$3\in S$和$6\in S$。不过，命题$2\in S$为假，说其他任何内容是S的成员的命题也都为假。

集合还能以其他集合作为成员。例如，设$T=\{\{1,2\},3,\varnothing\}$。那么$T$就有3个成员。第一个成员是集合$\{1,2\}$，也就是说，含有1和2作为成员的集合。第二个成员是整数3。第三个成员是空集。下列命题是真命题：$\{1,2\}\in T$，$3\in T$，以及$\varnothing\in T$。不过，$1\in T$为假。也就是说，1是T的成员的成员，但这不意味着1是T本身的成员。

7.2.1　原子

在正式的集合论中，除了集合别无他物。不过，在非正式的集合论中，以及在基于集合的数据结构和算法中，可以放心地假设存在某些原子。原子是非集合元素。原子可以是集合的成员，但没有什么可以是原子的成员。谨记，空集就像原子那样是没有成员的。不过，空集是集合，而不是原子。

我们一般会假设整数和小写字母都是原子。在谈论数据结构时，使用复杂的数据类型作为原子的类型通常是很方便的。因此，原子可以是看上去不那么像“原子”的结构体或数组。

集合与表

虽然表的表示法$(x_1, x_2, \cdots, x_n)$与集合的表示法$\{x_1, x_2, \cdots, x_n\}$非常相似，但它们之间存在很大区别。首先，集合中元素的次序是无关紧要的。写为$\{1,2\}$的集合也可以写作$\{2,1\}$。相反，表$(1,2)$与表$(2,1)$就不是一回事。

其次，表中元素是可以重复的。例如，表$(1,2,2)$有3个元素，第一个是1，第二个是2，第三个也是2。集合$\{1,2,2\}$是不存在的。元素（比如这里的2）作为成员在集合中出现的次数不能超过一次。上述集合要有意义的话，它就与$\{1,2\}$或$\{2,1\}$（也就是只含有1和2这两个成员，不含其他成员的集合）相同。

有时候会提到多重集或无序单位组（bag），就是允许其中元素出现多次的集合。例如，我们可以说出现一次1和两次2的多重集。不过多重集与表是不同的，因为多重集中的元素也是没有次序的。

7.2.2　通过抽象对集合的定义

枚举集合的成员不是定义集合的唯一方式。通常，更方便的做法是，从某集合S与元素的某属性P开始，然后定义S中具有属性P的元素为集合。这一操作对应的表示法称为抽象，就是

$$\{x|x\in S\text{且}P(x)\}$$

或者说“S中元素x的集合都是具有属性P的x”。

上述表达式称为集合形成法（set former）。集合形成法中的变量x是对应某一表达式的，我们也可以用

$$\{y|y\in S\text{且}P(y)\}$$

来表示同一集合。

✦ 示例 7.2

设S是示例7.1中的集合{1,3,6}。设$P(x)$是属性“x为奇数”，则

$$\{x|x\in S\text{且}x\text{为奇数}\}$$

是定义集合{1,3}的另一种方式。也就是说，我们接受S中的元素1和3，因为它们是奇数，而6不是奇数，所以我们拒绝了它。

再举个例子，考虑源自示例7.1的集合$T=\{\{1,2\},3,\varnothing\}$，那么

$$\{A|A\in T\text{且}A\text{为集合}\}$$

就表示集合$\{\{1,2\},\varnothing\}$。

7.2.3 集合的相等性

一定不能将集合的实际组成与其表示形式相混淆。两个集合*相等*，也就是说，如果它们刚好有相同的成员，那么它们其实是相同的集合。因此，大多数集合有很多不同的表示方式，包括以某种次序直接枚举集合中的元素，以及使用抽象的表示。

✦ 示例 7.3

集合{1，2}是有且只有1和2这两个成员的集合。我们可以按照任一次序表示这两个元素，所以{1,2} = {2,1}。还有其他很多通过抽象表示该集合的方式，例如

$$\{x|x\in\{1,2,3\}\text{且 }x<3\}$$

就等于集合{1,2}。

7.2.4 无限集

我们不介意假设集合都是*有限的*，也就是说，存在某个整数n，使得集合刚好具有n个成员。例如，集合{1,3,6}具有3个成员。还有一些集合是*无限的*，意味着没有具体的整数能表示该集合中元素的个数。我们熟悉的无限集包括下列几种。

(1) **N**，非负整数集。

(2) **Z**，整数集。

(3) **R**，实数集。

(4) **C**，复数集。

通过抽象，可以根据这些集合创建其他的有限集。

✦ 示例 7.4

集合形成法

$$\{x|x\in\mathbf{Z}\text{且 }x<3\}$$

表示由所有负整数以及0、1和2组成的集合，而集合形成法

$$\{x|x\in\mathbf{Z}\text{且 }\sqrt{x}\in\mathbf{Z}\}$$

表示的是完全平方的整数集合，也就是{0,1,4,9,16,…}。

再看一个例子，设$P(x)$是“x为质数”（即$x>1$，且x只能被1和它本身整除）这一属性。那么质数集就可以表示为

$$\{x|x\in\mathbf{N}\text{且}P(x)\}$$

这一表达式表示的是无限集{2,3,5, 7,11,…}。

无限集有一些微妙而有趣的属性我们将在7.11节中再来讨论这一问题。

7.2.5　习题

(1) 集合$\{\{a,b\},\{a\},\{b,c\}\}$的成员都有哪些？

(2) 写出以下集合的集合形成法表达式。

(a) 大于1000的整数集合。

(b) 偶整数的集合。

(3) 用两种不同的表示方式分别表示下列各集合，一种使用抽象，另一种不使用抽象。

(a) $\{a, b, c\}$

(b) $\{0, 1, 5\}$

罗素悖论

有人也许想知道，为什么抽象操作要求我们指明另一个集合，然后必须从该集合中选出构成新集合的元素。为什么不能直接使用$\{x|\ P(x)\ \}$这样的表达式，例如，用

$$\{x|x\text{是蓝色的}\}$$

来定义由所有蓝色事物组成的集合呢？原因在于，如果用这种概括方式来定义集合，就会让我们陷入一种由数学家伯特兰·罗素（Bertrand Russell）发现的名为罗素悖论（Russell Paradox）的逻辑矛盾之中。我们在听说镇上的理发师只给不自己剃胡子的人剃胡子时，就已经接触到这一悖论了。如果他给自己剃胡子，就不该给自己剃胡子；而如果他不给自己剃胡子，就可以给自己剃胡子。引发这种矛盾的原因是“只给不自己剃胡子的人剃胡子”这一说法，尽管看起来很合理，但其实是说不通的。

要理解罗素悖论是如何关系到集合的，先假设可以用$\{x|P(x)\}$的形式对任意属性P定义集合。接着设属性$P(x)$是“x不是x的成员”。也就是说，设P属性在集合x不是其本身成员的情况下适用于该集合。令S为集合

$$S = \{x|x\text{不是}x\text{的成员}\}$$

现在问，“S是否为它本身的成员？”

情况1：假设S不是S的成员。那么$P(S)$为真，S就是集合$\{x|x$不是x的成员$\}$的成员。不过该集合就是S，所以通过假设S不是它本身的成员，我们证明了S其实是它本身的成员。因此，不能有S不是它本身成员的结论。

情况2：假设S是它本身的成员。那么S就不是集合 $\{x|x$不是x的成员$\}$的成员。不过该集合也就是S，这样就得出了S不是它本身成员的结论。

因此，当我们假设$P(S)$为假时，就证明了它为真，而当我们假设$P(S)$为真时，我们又证明了它为假。因为不管怎样都会得出矛盾，这样就只能把责任归咎于这一表示方法。也就是说，真正的问题在于按照这样的方式定义集合S是行不通的。

罗素悖论另一个有趣的推论是假设存在“所有元素的集合”也是行不通的。如果存在这样的“全集”，比方说U，那么就可以说　$\{x|x \in U\text{且}x\text{不是}x\text{的成员}\}$

而且再次得到了罗素悖论。这样就不得不完全放弃抽象。但是抽象操作十分实用，不容放弃。

7.3　集合的运算

有一些操作集合的特殊运算，比如并集和交集。大家可能熟悉其中很多运算，但我们在此将回顾一些最重要的运算，在下一节中我们将讨论这些运算的一些实现。

7.3.1 并集、交集和差集

也许结合集合最常用的方式就是进行以下3种运算。

(1) 两个集合S和T的并集，记作$S \cup T$，表示含有集合S或集合T中或同在二者之中的元素的集合。

(2) 两个集合S和T的交集，记作$S \cap T$，表示含有同在集合S和集合T中的元素的集合。

(3) 两个集合S和T的差集，记作$S - T$，表示含有在集合S中但不在集合T中的元素的集合。

✦ 示例 7.5

设S是集合$\{1,2,3\}$，T是集合$\{3,4,5\}$,那么

$S \cup T = \{1,2,3,4,5\}$，$S \cap T = \{3\}$，而$S - T = \{1,2\}$。

也就是说，$S \cup T$包含了出现在S或T中的所有元素。虽然3同时出现在S和T中，但是$S \cup T$中当然只能出现一个3，因为元素在一个集合中不能出现多次。$S \cap T$只含3，因为没有其他元素同时出现在S和T中。最后$S - T$含有1和2，因为这两个元素出现在S中而未出现在T中。元素3没有出现在$S - T$中，因为虽然它在S中出现了，但它也出现在T中了。

当集合S和T是概率空间中的事件时，并集、交集和差集就有了一层含义。$S \cup T$就是S发生或T发生（或都发生）的事件。$S \cap T$就是S和T都发生的事件。$S - T$是S发生但T不发生的事件。不过，如果S是表示整个概率空间的集合，那么$S - T$就是"T不发生"这一事件，也就是T的补集。

7.3.2 文氏图

将涉及集合的运算看作称为文氏图（Venn diagrams）的图片通常是很有用的。图7-1中的文氏图表示S和T这两个集合，它们在图中表示为两个椭圆。这两个椭圆将整个平面分为4个区域，我们分别用数字1到4标记这4个区域。

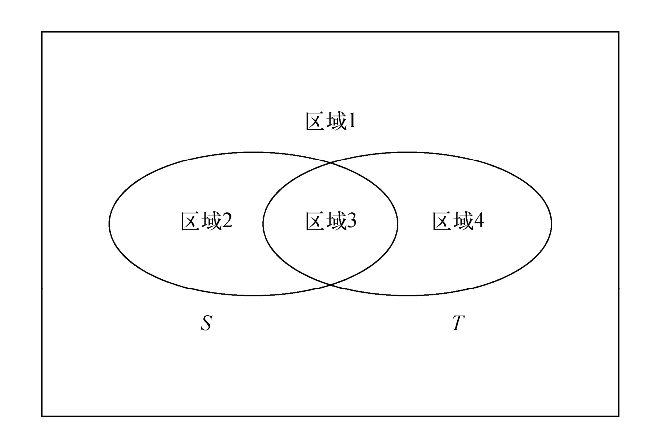


图7-1 表示对应基本集合运算的文氏图的区域

(1) 区域1表示既不在S中也不在T中的元素。

(2) 区域2表示$S - T$，那些在S中但不在T中的元素。

(3) 区域3表示$S \cap T$，那些既在S中也在T中的元素。

(4) 区域4表示$T - S$，那些在T中但不在S中的元素。

(5) 区域2、3、4结合在一起表示$S \cup T$，那些在S中或在T中，或同在二者之中的元素。

代数是什么？

可以想到，术语“代数”指的是解决单词问题，求出多项式的根，以及高中代数课程中涵盖的其他问题。不过，对数学家来说，术语“代数”指的是存在可用于构建表达式的操作数和运算符的任一种系统。为了让代数变得有趣而实用，通常会有一些特殊常量和法则，让我们可以将一个表达式变形为另一个“等价的”表达式。

最常见的代数的操作数是整数、实数或是复数，或是表示这些类型中某一种类型的值的变量，而运算符则是普通算术运算符——加号、减号、乘号、除号。常数0和1是特殊的，而且满足$x+0=x$这样的法则。在处理算术表达式时，可以使用诸如分配律这样的法则，让我们用$a\times(b+c)$这样的等价表达式来替代形如$a\times b+a+c$的表达式。请注意，通过这种变形，可以减少一次算术运算。对表达式进行这种代数变换的目的通常是找出与原表达式等价，但求值所需时间更少的表达式。

纵观全书，我们会遇到各种类型的代数。8.7节介绍了关系代数，是对我们在此讨论的集合代数的一般化；10.5节谈论了描述字符串模式的正则表达式代数；12.8节介绍了逻辑类型的布尔代数。

尽管我们已经说明了图7-1中的区域1具有有限的范围，不过应该记住的是，该区域表示的是S和T之外的一切。因此，该区域并非集合。如果该区域是集合，那么将其与S和T进行并集运算，就会得到“全集”，而根据罗素悖论可知这种“全集”是不存在的。不过，通常可以将不在文氏图明确表示的任一集合中的元素画为一个区域，就像我们在图7-1中所做的。

7.3.3　并集、交集和差集的代数法则

人们可以仿照诸如+和*这样的算术运算代数来定义集合代数，在集合代数中，运算符就是并集、交集和差集，而操作数就是集合或表示集合的变量。一旦可以构建$R\cup((S\cap T)-U)$这样的复杂表达式，就可以询问两个表达式是否等价。也就是说，不管用什么集合替换作为操作数的变量，它们总是表示相同的集合。通过将一个表达式替换为等价的表达式，有时能简化涉及集合的表达式，使其能更高效地求值。

接下来的内容中，我们将列出用于并集、交集和差集的最重要的代数法则，也就是断言一个表达式与另一个表达式等价的命题。符号≡用于表示表达式的相等性。

在多种代数法则中，一方面是在并集、交集和差集之间有着一种相似性，另一方面是与整数的加法、乘法和减法相似。不过，我们将指出那些与普通算术不存在相似性的法则。

(a) 并集的交换律：$(S\cup T)\equiv(T\cup S)$。也就是说，在并集运算中，两个集合中哪个出现在前面都是没关系的。这一法则成立的原因很简单。如果x在S中，或x在T中，或同在两者之中，就有元素x在$S\cup T$中。而这正好就是x在$T\cup S$中所要满足的条件。

(b) 并集的结合律：$(S\cup(T\cup R))\equiv((S\cup T)\cup R)$。也就是说，3个集合的并集既可以写为首先求前两个集合的并集，也可以写为首先求后两个集合的并集，不管哪种情况，结果都是一样的。我们可以像验证交换律那样，通过论证当且仅当元素在右边的集合中时才在左边的集合中，从而验证结合律。直观的理由就是两个集合中含有的，都正好是那些出现在S、T或R中，或任意两者之中，或同在三者之中的那些元素。

并集的交换律和结合律一起告诉我们，可以按照任意次序为一系列集合求并集。结果总是同一个元素集合，也就是出现在需要求并集的一个或多个集合中的那些元素组成的集合。这一论证就像我们在2.4节中表示加法时用到的，而加法运算中也存在交换律和结合律。那时我们证明过用所有方法组合求和算式都会得到相同的结果。

(c) 交集的交换律：$(S \cap T) \equiv (T \cap S)$。直觉上讲，元素$x$在集合$S \cap T$和$T \cap S$中刚好是在相同的情况下，也就是当$x$在$S$中而且$x$在$T$中时。

(d) 交集的结合律：$(S \cap (T \cap R)) \equiv (S \cap T) \cap R)$。直觉上讲，元素$x$在所述的这两个集合中，刚好是当元素$x$同在$S$、$T$和$R$这3个集合中时。就像加法或并集运算那样，对任意一些集合的交集运算也可以按照我们选择的方式进行组合，而且结果都是相同的，特别要指出的是，结果就是同时出现在所有集合中的那些元素。

(e) 交集对并集的分配律：就像我们所了解的乘法对加法的分配律，即$a \times (b+c) = a \times b + a \times c$那样，法则

$$(S \cap (T \cup R)) \equiv ((S \cap T) \cup (S \cap R))$$

对集合而言也成立。直觉上讲，元素x要分别在这两个集合中，刚好是当x在S中，并且至少在T和R其中一个之中时。同样，利用并集和交集的交换律，可以从右边起分配交集，就像

$$((T \cup R) \cap S) \equiv ((T \cap S) \cup (R \cap S))$$

(f) 并集对交集的分配律：同样，

$$(S \cup (T \cap R)) \equiv ((S \cup T) \cap (S \cup R))$$

是成立的。左右两边都是包含在S中或同时在T和R中的元素x的集合。请注意，将并集运算替换为加法，并将交集运算替代为乘法，这样形成的算术运算法则为假；也就是说，$a + b \times c$ 在一般情况下是不等于$(a+b) \times (a+c)$的。这一法则就是集合运算与算术运算相似性被打破的情况之一。就像在(e)法则中那样，我们可以利用并集的交换律得到等价法则

$$((T \cap R) \cup S) \equiv ((T \cup S) \cap (R \cup S))$$

✦ 示例 7.6

设$S = \{1,2,3\}$，$T = \{3,4,5\}$，$R = \{1,4,6\}$。那么

$$\begin{aligned} S \cup (T \cap R) &= \{1,2,3\} \cup (\{3,4,5\} \cap \{1,4,6\}) \\ &= \{1,2,3\} \cup \{4\} \\ &= \{1,2,3,4\} \end{aligned}$$

另一方面

$$\begin{aligned} (S \cup T) \cap (S \cup R) &= (\{1,2,3\} \cup \{3,4,5\}) \cap (\{1,2,3\} \cup \{1,4,6\}) \\ &= \{1,2,3,4,5\} \cap \{1,2,3,4,6\} \\ &= \{1,2,3,4\} \end{aligned}$$

因此，并集对交集的分配律在这种情况下是成立的。当然，这并没有证明这一法则在一般情况下是成立的，不过我们在规则(f)中给出的直觉论证应该是有说服力的。

(g) 并集和差集的结合律：$(S-(T \cup R)) \equiv ((S-T) - R)$。两边所包含的元素$x$，刚好都是在$S$中，而既不在$T$中，也不在$R$中。请注意，这条法则与算术法则$a-(b+c) = (a-b)-c$是相似的。

(h) 差集对并集的分配律：$((S \cup T) - R) \equiv ((S-T) \cup (T-R))$。两边集合中的元素$x$，都不在$R$中，但要么在$S$中，或在$T$中，或同在$S$和$T$两者之中。这一法则在算术运算中并没有相似法则，$(a+b)-c = (a-c)+(b-c)$是不成立的，除非$c=0$。

(i) 空集是并集的单位元(identity): 也就是说, $(S\cup\varnothing)\equiv S$, 而根据并集的交换律, 有 $(\varnothing\cup S)\equiv S$。粗略地讲，只有在元素$x$在$S$中时，它才可能在$S\cup\varnothing$中，因为$x$不可能在$\varnothing$中。

请注意，交集是没有单位元的。可以想象一下，包含“所有元素”的“集合”可以作为交集的单位元，因为集合S与该“集合”的交集肯定是S。不过，正如在介绍罗素悖论时提过的，不可能存在“具有所有元素的集合”。

(j) 并集的幂等律。将某运算符应用到同一个值的两个副本上，如果得到的结果还是该值，就说该运算符是幂等的。可知有$(S\cup S)\equiv S$。也就是说，在$(S\cup S)$中的元素x，刚好也就是在S中的元素x。该法则在算术运算中也没有相似法则，因为$(S\cap S)\equiv S$一般情况下不等于a。

(k) 交集的幂等律。同样地，我们有$(S\cap S)\equiv S$。

还有一些与对空集的运算有关的法则，如下所示。

(l) $(S-S)\equiv\varnothing$。

(m) $(\varnothing-S)\equiv\varnothing$。

(n) $(\varnothing\cap S)\equiv\varnothing$，而且根据交集的交换律，有$(S\cap\varnothing)\equiv\varnothing$。

7.3.4　利用文氏图证明相等性

图7-2用文氏图表示了交集对并集的分配律。该图展示了3个集合S、T和R，它们将平面分为8个区域，分别用数字1到8标记。这些区域对应着元素与这3个集合间8种可能存在的（在或不在集合中）关系。

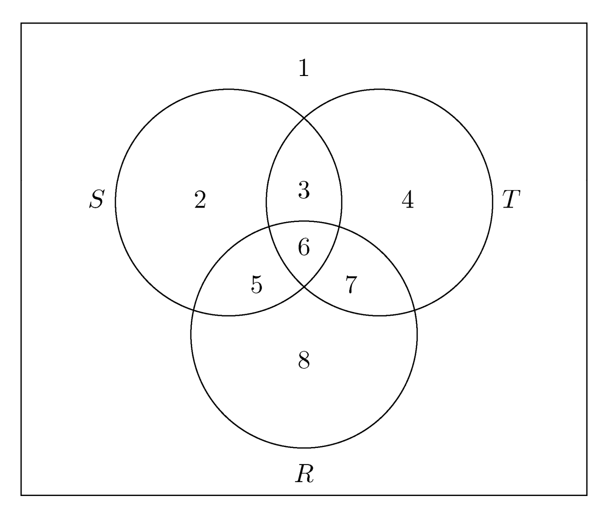


图7-2　表示交集对并集分配律的文氏图：$S\cap(T\cup R)$由区域3、5和6组成，$(S\cap T)\cup(S\cap R)$也是由这些区域组成

我们可以利用该图记录各子表达式的值。例如，$T\cup R$是区域3、4、5、6、7、8。因为S是区域2、3、5、6，所以$S\cap(T\cup R)$就是区域3、5、6。同样，$S\cap T$是区域3、6，而$S\cap R$是区域5、6。这样一来，$(S\cap T)\cup(S\cap R)$是同样的区域3、5、6，这就证明了

$$\big(S\cap(T\cup R)\big)\equiv\big((S\cap T)\cup(S\cap R)\big)$$

一般来说，通过从每个区域考虑一个具有代表性的元素，并验证它要么同在等式两边描述的集合中，要么都不在这两个集合中，我们可以证明相等性。这一方法与我们在第12章中证明命题逻辑的代数法则时用到的真值表方法是非常近似的。

7.3.5　利用变形证明相等性

另一种证明两个表达式相等的方式，是使用我们见过的代数法则将一个表达式变形为另一个表达式。我们将在第12章中更为正式地讲解如何处理表达式，现在只要注意到可以进行下列操作即可。

(1) 用任一表达式替换相等关系中的任一变量，要替换所有在该相等关系中出现的该变量。相等关系仍然成立。

(2) 设E是某相等关系中的子表达式，用已知与E等价的表达式F替代E。相等关系仍然成立。

此外，还可以直接写下任何表述为法则的相等关系，并假设这种相等关系是成立的。

✦　示例 7.7

我们要证明相等关系$\big(S-(S\cup R)\big)\equiv\varnothing$。首先使用法则(g)，并集和差集的结合律，也就是

$$\big(S-(T\cup R)\big)\equiv\big((S-T)-R\big)$$

我们用S替换相等关系中出现的两个T，就得到新的相等关系

$$\big(S-(S\cup R)\big)\equiv\big((S-S)-R\big)$$

根据规则(l)，$(S-S)\equiv\varnothing$。因此，可以用$\varnothing$替换上面的$(S-S)$，得到

$$\big(S-(S\cup R)\big)\equiv(\varnothing-R)$$

用R替代法则(m)中的S，就有$\varnothing-R\equiv\varnothing$。因此可用$\varnothing$替换$\varnothing-R$，从而得到$\big(S-(S\cup R)\big)\equiv\varnothing$。

7.3.6　子集关系

集合间也有一系列的比较运算符，它们与数字间的比较运算符相似。如果S和T都是集合，当S中的各成员也都是T的成员时，就说$S\subseteq T$。我们可以用多种方式表示这种关系："S是T的子集"、"T是S的超集"、"S包含于T"、"T包含S"。

如果$S\subseteq T$，而且T中至少有一个元素不是S中的成员，就说$S\subset T$。这一关系可以说成是"S是T的真子集"、"T是S的真超集"、"S真包含于T"、"T真包含S"。

就像"小于"关系那样，也可以反转这种比较的方向，$S\supset T$等同于$T\subset S$，而$S\subseteq T$等同于$T\subseteq S$。

✦　示例 7.8

以下比较关系都是成立的。

(1) $\{1,2\}\subseteq\{1,2,3\}$

(2) $\{1,2\}\subset\{1,2,3\}$

(3) $\{1,2\}\subseteq\{1,2\}$

请注意，集合永远是自身的子集，但从不可能是自身的真子集，所以$\{1,2\}\subset\{1,2\}$是不成立的。

还有一些涉及子集运算符和我们见过的其他运算符的代数法则，下面列出了一些。

(o)　对任一集合S，$\varnothing\subseteq S$

(p)　如果$S\subseteq T$，那么

　(i)　$(S\cup R)\equiv T$，

　(ii)　$(S\cap R)\equiv S$，且

　(iii)　$(S-T)\equiv\varnothing$。

7.3.7　通过证明则包含关系对相等性加以证明

当且仅当 $S \subseteq T$ 且 $T \subseteq S$ 时，则有两个集合S和T相等。因为，如果S中的每个元素都是T的元素，而且反之亦然，那么S和T刚好有着相同的成员，因此这两者就是相等的。反过来讲，如果S和T有着相同的成员，那么肯定有 $S \subseteq T$ 和 $T \subseteq S$ 都成立。这一规则与这样一条算术规则类似，就是当且仅当 $a \leqslant b$ 和 $b \leqslant a$ 都成立时有 $a = b$。

通过证明某一集合中的每个元素都包含在另一个集合中，可以证明两个表达式E和F的相等性。也就是说，我们

(1) 考虑E中的任意元素x，并证明它也在F中，然后

(2) 考虑F中的任意元素x，并证明它也在E中。

请注意，要证明$E \equiv F$，两个方向的证明都是必要的。

✦　示例 7.9

现在来证明并集和差集的结合律，

$$\big(S-(T\cup R)\big) \equiv \big((S-T)-R\big)$$

首先假设x在左边的表达式中，一系列的步骤如图7-3所示。请注意，在第(4)和第(5)步中，我们反向使用了并集的定义。也就是说，(3)告诉我们x不在 $T\cup R$ 中。如果x在T中，(3)就是不对的，所以可以得出x不在T中的结论。同样，x不在R中。

	步　骤	原　因
1)	x在 $S-(T\cup R)$ 中	给定
2)	x在S中	−的定义，以及(1)
3)	x不在 $T\cup R$ 中	−的定义，以及(1)
4)	x不在T中	$\cup$ 的定义，以及(3)
5)	x不在R中	$\cup$ 的定义，以及(3)
6)	x在$S-T$中	−的定义，以及(2)和(4)
7)	x在 $(S-T)-R$ 中	−的定义，以及(6)和(5)

图7-3　并集和差集的结合律的一半证明

这还没完，我们必须从假设x在 $(S-T)-R$ 中开始，并证明它在 $S-(T\cup R)$ 中。证明步骤如图7-4所示。

	步　骤	原　因
1)	x在$(S-T)-R$中	给定
2)	x在 $S-T$ 中	−的定义，以及(1)
3)	x不在R中	−的定义，以及(1)
4)	x在S中	−的定义，以及(2)
5)	x不在T中	−的定义，以及(2)
6)	x不在 $T\cup R$ 中	$\cup$ 的定义，以及(3)和(5)
7)	x在 $S-(T\cup R)$ 中	的定义，以及(4)和(6)

图7-4　并集和差集的结合律的另一半证明

✦ 示例 7.10

再举个例子，证明(p)法则的一部分，如果 $S \subseteq T$ ，那么 $S \cup R \equiv T$ 。首先假设x在 $S \cup T$ 中。我们根据并集的定义可知，只可能存在下列情况之一

(1) x在S中；

(2) x在T中。

在情况(1)中，因为假设有 $S \subseteq T$ ，所以可知x在T中。在情况(2)中，直接就可以看出x在T中。因此，在任一情况下x都在T中，这样就完成了证明的第一半——命题 $(S \cup T) \subseteq T$ 。

再来假设x在T中。那么根据并集的定义就有x在 $S \cup T$ 中。因此，$T \subseteq (S \cup T)$ ，这就是证明的第二半。这样就可以得出，如果 $S \subseteq T$ ，那么 $S \cup T \equiv T$ 。

7.3.8 集合的幂集

如果S是任一集合，那么S的幂集就是指由S的所有子集组成的集合。我们将用P(S)表示S的幂集，虽然有时也会使用2^S这样的表示法。

✦ 示例 7.11

设 $S = \{1,2,3\}$ 。那么

$$\mathbf{P}(S) = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$$

也就是说，P(S)是含有8个成员的集合，每个成员本身都是一个集合。空集也在P(S)中，因为显然有 $\varnothing \subseteq S$ 。单元素集——由S中的一个元素构成的集合，即$\{1\}$、$\{2\}$、$\{3\}$——也在P(S)中。同样，从3个成员中任选两个组成的3个集合在P(S)中，而S本身也是P(S)的成员。

再举一个例子，P($\varnothing$)= {$\varnothing$}因为$\varnothing \subseteq S$ ，而除空集之外，没有任何集合S可以满足 $S \subseteq \varnothing$ 。请注意，{$\varnothing$}是包含空集的集合，它和空集是不一样的。特别要指出的是，{$\varnothing$}含有一个成员，也就是$\varnothing$ ，而空集是不含任何成员的。

7.3.9 幂集的大小

如果S有n个成员，那么P(S)有2^n个成员。在示例7.11中，我们看到有3个成员的集合的幂集共有2^3= 8个成员。此外，2^0= 1，而且我们看到，包含0个元素的空集的幂集刚好有1个元素。

设 $S = \{a_1, a_2, \cdots, a_n\}$ ，其中 $a_1, a_2, \cdots, a_n$ 是任意n个元素。现在要通过对n的归纳证明，P(S)有2^n个成员。

依据。如果 $n = 0$ ，那么S就是$\varnothing$ 。我们之前已经得出**P**($\varnothing$)有一个成员的结论。因为2^0= 1，所以我们证明了依据情况。

归纳。假设当 $S = \{a_1, a_2, \cdots, a_n\}$ 时，P(S)有2^n个成员。设a_{n+1}是一个不同于S中任一元素的新元素，并设 $T = S \cup \{a_{n+1}\}$ ，该集合是个具有 $n+1$ 个元素的集合。现在，T的子集要么含有a_{n+1}这一成员，要么不含这一成员。我们来依次考虑这两种情况。

(1) 不包含a_{n+1}的T的子集，也是S的子集，因此在P(S)中。而根据归纳假设，正好有2^n个这样的集合。

(2) 如果R是包含a_{n+1}的T的子集，设 $Q = R - \{a_{n+1}\}$ ，也就是说，Q是将a_{n+1}删除后的R。那么Q是S的子集。根据归纳假设，刚好有2^n个可能存在的集合Q，而每一个都与唯一的集合R（也就是 $Q \cup \{a_{n+1}\}$ ）对应。

我们得出T刚好有2×2^n，也就是2^{n+1}个子集，其中有一半也是S的子集，而另一半则是由S的各子集分别加上新元素a_{n+1}形成的。因此，归纳步骤得到证明，给定任一具有n个元素的集合S都有2^n个子集的条件，就证明了具有$n+1$个元素的任一集合T都有2^{n+1}个子集。

7.3.10　习题

(1) 在图7-2中，我们证明了两个表达式对应着区域集合{3,5,6}。不过，每个区域都可以表示为涉及S、T和R，以及并集、交集和差集运算符的表达式。写出对应以下各区域的两种不同的表达式。

(a) 区域6。

(b) 区域2和区域8。

(c) 区域2、区域4和区域8。

(2) 使用文氏图证明以下代数法则。对于相等关系中涉及的每个子表达式，指出它所表示的区域集合。

(a) $\big(S\cup(T\cap R)\big)\equiv\big((S\cup T)\cap(S\cup R)\big)$

(b) $\big(S\cup T)-R\big)\equiv\big((S-R)\cup(T-R)\big)$

(c) $\big(S-(T\cup R)\big)\equiv\big((S-T)-R\big)$

(3) 通过证明每一边对另一边的包含关系，证明习题(2)中的各相等关系。

(4) 假设$S\subseteq T$，通过证明两边互为另一边的子集，证明如下相等关系：

(a) $(S\cup T)\equiv S$

(b) $(S-T)\equiv\varnothing$

(5) * 假设没有集合是其他集合的子集，那么包含n个集合的文氏图可将平面分割成多少个区域？假设n个集合中有一个是另一个的子集，但没有其他的包含关系。那么有些区域就将是空的。例如，在图7-1中，如果$S\subseteq T$，那么区域2就将为空，因为没有在S中而不在T中的元素。一般而言，共有多少个非空区域？

(6) 证明，如果$S\subseteq T$，那么$\boldsymbol{P}(S)\subseteq\boldsymbol{P}(T)$。

(7) * 在C语言中，我们可以用元素为链表表头的链表来表示成员为集合的集合S，这些元素对应的链表都表示S的成员之一。编写C语言程序，接受表示集合的元素构成的表，即表中元素各不相同的表，并返回给定集合的幂集。大家编写的程序的运行时间是多少？提示：利用对“含n个元素的集合的幂集中有2^n个成员”这一命题的归纳证明，得出创建幂集的递归算法。如果脑筋灵活点，就会使用同一个表作为若干集合的相同部分，从而避免复制表示幂集成员的表，这样既能节省时间，又能节省空间。

(8) 证明

(a) $\boldsymbol{P}(S)\cup\boldsymbol{P}(T)\subseteq\boldsymbol{P}(S\cup T)$

(b) $\boldsymbol{P}(S\cap T)\subseteq\boldsymbol{P}(S)\cap\boldsymbol{P}(T)$

如果将这里的包含关系替换为相等关系，那(a)或(b)是否还成立？

(9) $\boldsymbol{P}\Big(\boldsymbol{P}\big(\boldsymbol{P}(\varnothing)\big)\Big)$是什么？

(10) * 如果从$\varnothing$开始，应用幂集运算符n次，那么得到的集合中有多少个成员？例如，习题(9)就是$n=3$的情况。

7.4　集合的链表实现

我们已经在6.4节中看到过如何用链表数据结构实现词典操作插入、删除和查找。同时还看到，如果集合有n个元素，那么这些操作的期望运行时间都是$O(n)$。这一运行时间不如5.8节中使用平衡二叉查找树实现词典操作平均为$O(\log n)$的运行时间那样理想。另一方面，正如在7.6

节中将要看到的，用来表示词典的散列表数据结构是以词典的链表表示为基础的，而它一般要比二叉查找树快。

7.4.1　并集、交集和差集

尽管具体技巧与我们应用在词典操作上的有所不同，但使用链表数据结构还是对诸如并集这样的基本集合运算有利的。特别要说明的是，为表排序可以显著改善并集、交集和差集运算的运行时间。而我们在6.4节中看到的，排序只能对词典操作的运行时间带来比较小的改善。

首先，看看在用未排序表表示集合时会出现什么问题。在这种情况下，要对大小分别为n和m的集合进行并集、交集或差集运算，就需要$O(mn)$的时间。例如，要创建表示集合S与集合T的并集的表U，首先要将表示S的表复制到一开始为空表的U中。然后对T中的各个元素加以检验，看看它们是否也在S中。如果不在，就将该元素添加到U中。图7-5简要描述了这一思路。

```
(1)    copy S to U;
(2)    for (each x in T)
(3)        if (!lookup(x, S))
(4)            insert(x, U);
```

图7-5　为用未排序表表示的集合求并集的伪代码概要

假设S含有n个成员，而T含有m个成员。那么第(1)行将S复制到U中的操作可以在$O(n)$时间内完成。如果从第(3)行得知x不在S中，那么只要执行第(4)行的插入即可。因为x只可以在表示T的表中出现一次，所以可知x还不在U中。因此，将x放在表示U的表的前端是没问题的，并且第(4)行可以在$O(1)$时间内完成。第(2)行至第(4)行的`for`循环要迭代m次，而且其循环体要花费$O(n)$的时间。因此，第(2)行至第(4)行的运行时间就是$O(mn)$，它主导了第(1)行的$O(n)$时间。

还有与之类似的实现交集和差集运算的算法，所花的时间也都是$O(mn)$。我们在此将这些算法留给读者来设计。

7.4.2　使用已排序表的并集、交集和差集

当表示集合的表已经排序时，执行并集、交集和差集运算就要快得多。其实，大家会发现，即便这些表一开始没有排过序，在执行这些集合运算之前先给表排序都是值得的。例如，考虑一下$S \cup T$的计算，其中S和T都是用已排序表表示的。这一过程就和2.8节的归并算法类似。区别之一在于，在当前位于两表开头位置的最小元素相同时，只需要给出该元素的一个副本即可，而不用像归并那样必须给出两个副本。另一个区别在于，我们不能从表示用来求并集的集合S和T的表中直接删除元素，因为不应该在构建S和T的并集时对S或T造成破坏。我们必须为所有元素创建副本，用以形成二者的并集。

假设类型LIST和CELL是像之前那样，通过宏

```
DefCell(int, CELL, LIST);
```

定义的。函数`setUnion`如图7-6所示。在第(1)行要利用辅助函数`assemble(x, L, M)`创建一个新单元，在第(2)行将元素x放入该单元，并在第(3)行调用`setUnion`求表L和M的并集。然后，`assemble`会返回对应x的单元，后面跟着对L和M应用`setUnion`后得到的表。请注意，`assemble`和`setUnion`这两个函数是相互递归的，每一个都会调用另一个。

函数`setUnion`会从两个给定的已排序表中选出最小的元素，并将选定的元素与两个表其

余的部分一起传给assemble。对setUnion来说有6种情况，具体取决于两个表中有没有一个为NULL，如果没有，就要看两个表中哪个表表头位置的元素先于另一个。

(1) 如果两个表都为NULL，setUnion就直接返回NULL，结束递归过程。这种情况就是图7-6中的第(5)行和第(6)行。

(2) 如果*L*为NULL而*M*不是，那么在第(7)行和第(8)行，通过从*M*中取出第一个元素，后面跟上NULL表与*M*尾部的"并集"，就组成了这两个表的并集。请注意，在这种情况下，对setUnion的成功调用会使*M*被复制下来。

(3) 如果*M*为NULL而*L*不是，那么在第(9)行和第(10)行，要完成的工作是相反的，用*L*的第一个元素合*L*的尾部组成答案。

(4) 如果*L*和*M*的第一个元素是相同的，那么在第(11)行和第(12)行，就创建该元素的一个副本，表示为L->element，加上*L*的尾部和*M*的尾部，一起构成答案。

(5) 如果*L*的第一个元素先于*M*，那么在第(13)行和第(14)行，我们会用该最小元素，*L*的尾部，以及整个表*M*一起组成答案。

(6) 对称地，在第(15)行和第(16)行，如果最小元素在*M*中，我们就用该元素、整个表*L*，以及*M*的尾部组成答案。

```
        LIST setUnion(LIST L, LIST M);
        LIST assemble(int x, LIST L, LIST M);

        /* 由assemble函数生成的表，其表头元素为x且
           尾部为表L和表M并集中所含元素 */

        LIST assemble(int x, LIST L, LIST M)
        {
            LIST first;

 (1)        first = (LIST) malloc(sizeof(struct CELL));
 (2)        first->element = x;
 (3)        first->next = setUnion(L, M);
 (4)        return first;
        }

        /* setUnion 返回的表是L和M的并集 */

        LIST setUnion(LIST L, LIST M)
        {
 (5)        if (L == NULL && M == NULL)
 (6)            return NULL;
 (7)        else if (L == NULL) /* M在这里不能为NULL */
 (8)            return assemble(M->element, NULL, M->next);
 (9)        else if (M == NULL) /* L在这里不能为NULL */
(10)            return assemble(L->element, L->next, NULL);
           /* 如果到了这里，L和M都不能为NULL */
(11)        else if (L->element == M->element)
(12)            return assemble(L->element, L->next, M->next);
(13)        else if (L->element < M->element)
(14)            return assemble(L->element, L->next, M);
(15)        else /* 这里有M->element < L->element */
(16)            return assemble(M->element, L, M->next);
        }
```

图7-6　为用已排序表表示的集合计算并集

✦ 示例 7.12

假设集合S是{1,3,6}，T是{5,3}。表示这两个集合的已排序表分别是 $L=(1,3,6)$ 和 $M=(3,5)$ 。调用`setUnion (L,M)`求并集。因为L的第一个元素是1，先于M的第一个元素3，所以情况(5)适用，因此我们用1，L的尾部，称其为L_1=(3,6)，以及M组成要计算的并集。函数`assemble(1,L,M)`会在第(3)行调用`setUnion(L,M)`，结果就是第一个元素1与等于并集的尾部组成的表。

对`setUnion`的这一调用是情况(4)，也就是两个开头元素相等的情况，这里都是3。因此，我们用元素3的一个副本，加上L_1的尾部和M的尾部，组成要计算的并集。这些尾部分别是只有元素6组成的L_2，以及只由元素5组成的M_1。接下来的调用是`setUnion(L2,M1)`，这是情况(6)的实例。因此我们将5加到并集中，并调用`setUnion(L2,NULL)`。这是情况(3)，为并集生成6，并调用`setUnion(NULL,NULL)`。这里就遇到了情况(1)，递归就终止了。对`setUnion`首次调用的结果就是表(1,3,5,6)。图7-7详细展示了这一套示例数据产生的调用与返回。

```
调用 setUnion((1,3,6),(3,5))
    调用 assemble(1,(3,6),(3,5))
        调用 setUnion((3,6),(3,5))
            调用 assemble(3,(6),(5))
                调用 setUnion((6),(5))
                    调用 assemble(5,(6),NULL)
                        调用 setUnion((6),NULL)
                            调用 assemble(6,NULL,NULL)
                                调用 setUnion(NULL,NULL)
                                返回 NULL
                            返回 (6)
                        返回 (6)
                    返回 (5,6)
                返回 (5,6)
            返回 (3,5,6)
        返回 (3,5,6)
    返回 (1,3,5,6)
返回 (1,3,5,6)
```

图7-7　示例7.12对应的调用合返回序列

请注意，`setUnion`生成的表总是已排序的。通过看到哪种情况适用，可以知道该算法为何起作用，表L或M中的各元素，要么通过成为对`assemble`调用中的第一个参数，从而被复制到输出中，要么留在作为参数被传递给对`setUnion`的递归调用的表中。

7.4.3　并集运算的运行时间

如果对分别具有n个和m个元素的集合调用`setUnion`，那么`setUnion`所花的时间就是$O(m+n)$。想明白为什么，要注意到对`assemble`的调用会花$O(1)$的时间为输出表创建一个单元，然后对剩下的表调用`setUnion`。因此，图7-6中对`assemble`的调用，可以视为要花$O(1)$的时间，再加上对长度之和为比L和M长度之和少1，或在情况(4)下比L和M长度之和少2的两个表调用`setUnion`所花的时间。此外，`setUnion`中的所有工作，除了对`assemble`的调用之外，所花时间都是$O(1)$。

多变量函数的大O

正如在6.9节中指出的，我们为单变量函数定义的大O概念自然也可以应用于多变量函数。如果存在常数c和a_1、⋯、a_k，使得对i=1、⋯、k，只要$x_i \geqslant a_i$，就有$f(x_1,\cdots,x_k) \leqslant cg(x_1,\cdots,x_k)$，就说$f(x_1,\cdots,x_k)$是$O\left(g(x_1,\cdots,x_k)\right)$。特别要说的是，虽然当$m$和$n$其中一个为0而另一个大于0时会有$m$+$n$大于$mn$，但通过选择常数$c$、$a_1$和$a_2$都等于1，仍然可以说$m$+$n$是$O(mn)$。

接下来，在总长度为m+n的两个表调用`setUnion`，这最多会造成m+n次对`setUnion`的递归调用，以及同样次数的对`assemble`的调用。除去递归调用花的时间，每次调用所花的时间为$O(1)$。因此，求并集所花的时间为$O(m+n)$，也就是说，与两个集合的大小之和成比例。

这一时间比为用未排序表表示的集合求并集所需的时间$O(mn)$要少。其实，如果表示集合的表是未排序的，可以在$O(n\log n + m\log m)$的时间内为这两个表排序，接着再对已排序的表求并集。因为$n\log n$主导了n，而$m\log m$主导了m，所以可以将排序与求并集的总时间支出表示为$O(n\log n + m\log m)$。这一表达式可能比$O(mn)$大，但只要n与m的值很接近，也就是说，只要两个集合的大小近似相同，它就比$O(mn)$小。因此，在求并集之前先排序是说得通的。

7.4.4　交集和差集

图7-6概述了求并集的算法思路，这一思路也适用于求交集和差集的运算：当集合用已排序表表示时，交集和差集运算也能以线性时间执行。对交集而言，只有当元素同时出现在两个集合中，也就是像之前的情况(4)那样时，才会把元素复制到输出中。如果有一个表为`NULL`，在交集中就不会有任何元素了，因此情况(1)、(2)、(3)就可以被替换为返回`NULL`的操作。在情况(4)中，我们将两个表表头的元素复制到交集中。而在情况(5)和情况(6)中，两个表的表头元素是不同的，这样较小的元素就不可能都出现在两个表中，因此就不用向交集中添加任何内容，而是要将较小的元素从其所在表中弹出，并对剩下部分求交集。

想知道为什么这样能行，可以举个例子，假设a是在表L的表头，b是在表M的表头，并且有$a<b$。那么a就不可能出现在已排序表M中，因此可以排除a同时出现在两个表中的可能。不过，b可能出现在表L中在a之后的某个位置，这样一来就仍然有可能用到来自M的b。因此，我们需要继续对L的尾部与整个表M求交集。相反，如果b小于a，就要对整个表L与M的尾部求交集。计算交集的C语言代码如图7-8所示。还需要修改`assemble`，用对`intersection`的调用替代对`setUnion`的调用。我们将这一修改以及为已排序表求差集的程序留作本节习题。

```
LIST intersection(LIST L, LIST M)
{
    if (L == NULL || M == NULL)
        return NULL;
    else if (L->element == M->element)
        return assemble(L->element, L->next, M->next);
    else if (L->element < M->element)
        return intersection(L->next, M);
    else /* 这里有 M->element < L->element */
        return intersection(L, M->next);
}
```

图7-8　为用已排序表表示的集合计算交集，这里需要新版本的`assemble`函数

7.4.5　习题

(1) 编写C语言程序，为用未排序表表示的集合求(a)并集；(b)交集；(c)差集。

(2) 修改图7-6中的程序，使其为用已排序表表示的集合求(a)交集；(b)差集。

(3) 图7-6中的`assemble`和`setUnion`函数不会改变原来的表，也就是说，它们会创建元素的副本，而非使用给定表本身的单元。大家能否通过在求并集的过程中销毁给定的表来简化程序？

(4) * 通过对作为参数给定的两个表的长度之和进行归纳，证明图7-6中的`setUnion`函数会返回给定表的并集。

(5) * 两个集合S和T的*对称差*是$(S-T)\cup(T-S)$，也就是说，刚好只出现在S或T其中一个之中的元素。编写程序，为用已排序表表示的两个集合求对称差。大家的程序应该像图7-6中那样只传递一次表，而不要调用求并集与求差集的程序。

(6) * 我们对图7-6中的程序进行了非正式的分析，论证了如果两个表的总长度为n，就会有$O(n)$次对`setUnion`和`assemble`的调用，而且每次调用花的时间是$O(1)$加上递归调用所花的时间。我们可以将这一论证过程正式化，设$T_U(n)$是`setUnion`对总长度为n的两个表的运行时间，$T_A(n)$是`assemble`对总长度为n的两个表的运行时间。分别写出T_U与T_A相互以对方定义自身的递归规则。进行替换，消去T_A，为T_U建立常规的递推关系。为该递推关系求解。这是否证明了`setUnion`所花时间为$O(n)$？

7.5　集合的特征向量实现

很多时候，我们遇到的一些集合是要称为“全集”[①]的某个小集合U的各子集。例如，扑克牌型就是由全部52张扑克牌组成的集合的子集。当我们关注的集合是某个小集合U的各子集时，存在一种比7.4节中讨论的表实现有效得多的集合实现方式。我们以某种方式为U中的元素排定次序，这样一来，U中的每个元素都可以与一个唯一的“位置”相关联，这一位置是从0到$n-1$的整数，其中n是U中元素的个数。

接着，给定一个包含于U的集合S，就可以用由0和1组成的特征向量表示S，其规则是，对U中的每个元素x，如果x在S中，对应x的位置上就是1，而如果x不在S中，对应的位置上就是0。

✦　示例 7.13

设U是一副扑克牌组成的集合。我们可以用任何方式为扑克牌排定次序，不过比较合理的模式是先按照它们的花色：梅花、方块、红桃和黑桃。然后，在同一花色中，按照A、2、3、⋯、10、J、Q、K这样的顺序排列。例如，梅花A的位置就是0，梅花K的位置是12，方块A的位置是13，而黑桃J的位置是49。红桃同花大顺（即红桃10、J、Q、K、A）是由以下特征向量表示的

$$0000000000000000000000000010000000011110000000000000$$

第一个1在位置26处，表示红桃A，而其他4个1则是在35到38这4个位置，它们分别表示红桃10、J、Q和K。

所有梅花花色的牌组成的集合是由以下特征向量表示的

$$1111111111111000000000000000000000000000000000000000$$

而所有花牌（即各花色的J、Q、K）组成的集合则是由以下特征向量表示的

$$0000000000111000000000011100000000001110000000000111$$

---

① 当然，U不可能是真正的全集，我们用罗素悖论论证过这种所有集合的集合是不存在的。

7.5.1　集合的数组实现

要表示某n元素全集各子集的特征向量，可以使用具有如下类型的布尔数组：

```
typedef BOOLEAN USET[n];
```

我们在1.6节中描述过BOOLEAN类型。要将对应位置i的元素插入到声明为USET类型的集合S中，只需要执行

```
S[i] = TRUE;
```

同样，要从S中删除对应位置i的元素，就要

```
S[i] = FALSE;
```

如果要查找该元素，只需返回值$S[i]$即可，该值就告诉了我们第i个元素是否出现在S中。

请注意，当集合用特征向量表示时，词典操作插入、删除和查找各需$O(1)$的时间。这一技巧的唯一缺点是，所有被表示的集合都必须是某个全集U的子集。此外，该全集必须很小，否则，数组就会变得很大，要存储数组就不方便了。事实上，因为我们通常一定要将表示集合的数组中所有元素初始化为TRUE或FALSE，而初始化U的任一子集（即便是$\varnothing$）所花的时间都肯定与U的大小成比例。如果U中有大量的元素，那么初始化集合所花的时间可能会主导所有其他操作的开销。

如果两个集合同为某n元素普通全集的子集，它们分别由特征向量S和T表示，要构成这两个集合的并集，可以定义另一个特征向量R来表示特征向量S和T的按位OR：

对$0\leqslant i\leqslant n$，`R[i] = S[i] || T[i]`

同样，要让R表示S和T的交集，就只要对S和T的特征向量按位AND：

对$0\leqslant i\leqslant n$，`R[i] = S[i] && T[i]`

最后，可以按照如下方式让R表示S和T的差集$S-T$：

对$0\leqslant i\leqslant n$，`R[i] = S[i] && !T[i]`

如果恰当地定义类型BOOLEAN，表示特征向量的数组及对这些数组执行的布尔运算都可以用C语言中的按位运算符实现。不过，这些代码都是与机器相关的，所以在这里不会展示任何细节。特征向量有一种可移植但更耗费空间的实现，可以用合适大小的int类型数组实现，而这是一种我们假设过的BOOLEAN类型的定义。

✦　示例 7.14

考虑一下苹果品种的集合。这里的全集由图7-9所示的6个品种构成，其排列次序表示了它们在特征向量中的位置。

	品　　种	颜　　色	成熟期
0)	美味（Delicious）	红	晚熟
1)	格兰尼・史密斯（Granny Smith）	绿	早熟
2)	格拉文施泰因（Gravenstein）	红	早熟
3)	乔纳森（Jonathan）	红	早熟
4)	旭苹果（McIntosh）	红	晚熟
5)	翠玉苹果（Pippin）	绿	晚熟

图7-9　某些苹果品种的特征

红苹果的集合是由特征向量

$$Red = 01110$$

表示的，而早熟苹果的集合是由特征向量

$$Early = 011100$$

表示的。因此，由红色或早熟的苹果品种构成的集合，即 $Red \cup Early$，是由特征向量111110表示的。请注意，这一向量为1的位置，是表示*Red*的特征向量101110中为1的位置，或是表示*Early*的特征向量011100中为1的位置，或是两者中都为1的位置。

通过在101110和011100都为1的位置放置1，可以得到表示 $Red \cap Early$（早熟红苹果的集合）的特征向量。得到的向量是001100，表示苹果品种的集合{格拉文施泰因・乔纳森}。而晚熟红苹果的集合，也就是

$$Red - Early$$

可以用向量100010表示。该集合为{美味，旭苹果}。

请注意，使用特征变量求并集、交集和差集所花的时间与向量的长度是成正比的。这一长度与待运算集合的大小没有直接关系，而是等于所选择全集的大小。如果待运算集合占据全集中相当可观的一部分元素，那么求并集、交集和差集的时间也和待运算集合的大小成比例。这一时间要优于已排序表的 $O(n \log n)$ 时间，且大大优于未排序表的 $O(n^2)$ 时间。不过，特征向量也有个缺点，假如所涉及集合的大小远小于全集的大小，这些运算的运行时间就要远大于所涉及集合的大小。

7.5.2 习题

(1) 给出如下扑克牌集合的特征向量。为了方便起见，大家可以使用0^k表示*k*个连续的0，用1^k表示*k*个连续的1。
  (a) 皮诺奇勒牌堆（使用4种花色的9、10、J、Q、K和A各两张）中的扑克牌。
  (b) 红色扑克牌。
  (c) 红桃J、黑桃J和红桃K。

(2) 使用按位运算符，编写C语言程序计算两个扑克牌集合的(a)并集；(b)差集，其中第一个集合是用单词*a*1和*a*2表示的，而第二个集合则是由*b*1和*b*2表示的。

(3) * 假设要表示元素包含于某小型全集*U*的无序单位组（多重集）。该如何将特征向量法推广到无序单位组的表示呢？说明要如何对这样表示的无序单位组执行(a)插入，(b)删除；(c)查找操作。请注意，无序单位组的*lookup*(*x*)返回的是*x*在无序单位组中出现的次数。

7.6 散列

在可以使用词典的特征向量表示时，我们可以直接访问表示元素的位置，也就是访问数组中以该元素的值为下标的位置。不过，正如前面提到过的，不能让全集的大小太大，否则数组长度就会超出计算机可用内存的容纳能力了。就算计算机内存能容纳这个数组，初始化数组所需的时间也太长了。例如，假设要存储真正的英文词典，并假设我们愿意忽略10个字母以上的单词。仍会有$26^{10}+26^9+ \cdots +26$个可能存在的单词，这大约是超过10^{14}个单词，每个可能的单词都需要数组的一个位置。

不过，不管什么时候，英语语言中一般只有100万个单词，所以之前所说的数组中只有一亿分之一的数据项为TRUE。我们也许可以缩减该书组，使得很多可能存在的单词共享一个数据项。

例如，假设指定头100万个单词存放在数组的第一个单元中，而接下来的100万个可能存在的单词存放在第二个单元中，以此类推，直到第100万个单元。这种安排有两个问题。

(1) 在单元中只放入TRUE已经不够了，因为我们没法知道这100万个可能的单词中到底有哪些实际出现在词典中，也不知道任意一组中是否有多个单词出现。

(2) 比方说，如果头100万个可能的单词包含了所有的短单词，就可以预期有超过平均数的词典内单词落入这一组可能存在的单词中。要注意到，我们的安排是数组单元数要和词典中的单词数相当，这样就可以预期平均每个单元要表示一个单词，但英语中肯定有好几千个单词是在第一组中的，这样就包含了所有不超过5个字母的单词，以及部分6个字母的单词。

要解决问题(1)，就需要在数组的每个单元中列出该组中出现在词典里的所有单词。也就是说，该数组单元成了容纳这些单词的链表的表头。要解决问题(2)，需要注意如何为潜在的单词分组。一定要合理分配各组中的元素，使得不大可能出现（虽然从不会不出现）某一组中有很多元素的情况，虽然这种情况不太可能不出现。请注意，如果在一组中有大量的元素，而且我们又用链表来表示组，那么在成员众多的组中查找元素就会非常缓慢。

7.6.1　散列表数据结构

我们现在已经从特征向量这种使用范围有限但很有价值的数据结构，演变到了对任意词典都很有用而且对很多其他用户来说也很实用的*散列表*数据结构。[①]散列表的词典操作速度平均可达 $O(1)$ 的水平，而且与构建词典所用全集大小没有关系。图7-10中展示了散列表的图片，不过，我们只给出了x所在的那一组对应的链表。

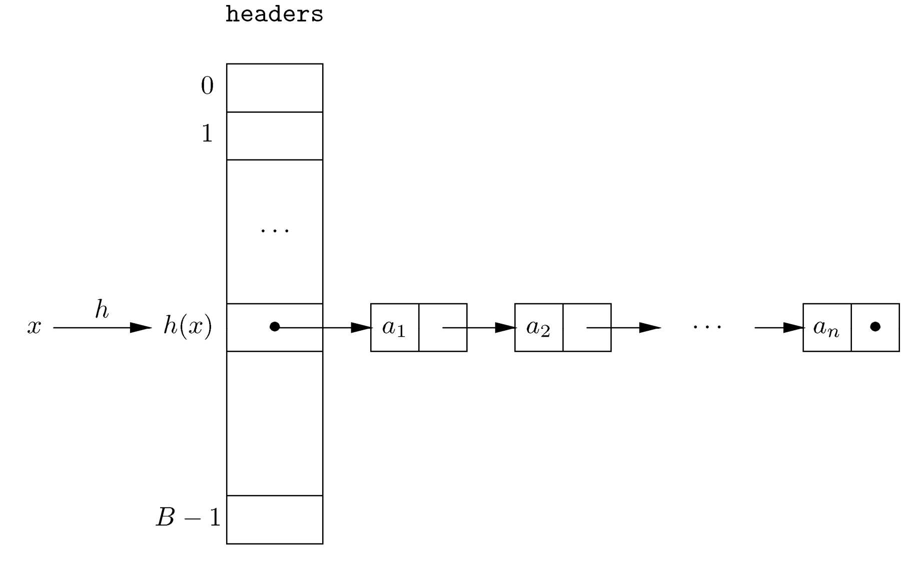


图7-10　散列表

*散列函数*接受元素x作为参数，并生成0到B−1之间的某个整数值，其中B是散列表中*散列表元*（bucket）的数量。值$h(x)$就是我们放置元素x的散列表元的位置。因此，这些散列表元与我们之前非正式讨论中谈论过的单词“组”是对应的，而散列函数是用来决定某个给定元

① 虽然有的情况下用特征向量也是可行的，但我们通常还是会优先选择用散列表来表示。

素应该属于哪个散列表元的。

使用何种散列函数更合适取决于元素的类型。例如

(1) 如果元素是整数，就可以令$h(x)$为$x\%B$，也就是x除以B的余数。这一数字总是在所要求的0到$B-1$这一范围内。

(2) 如果元素是字符串，就可以取元素$x = a_1a_2\cdots a_k$，其中每个a_i都是一个字符，并计算$y = a_1 + a_2 + \cdots + a_k$，因为在C语言中char类型是个小整数。这样，我们就得到与字符串x中所有字符等价的整数的和y。如果用y除以B，并取余数，就得到了在0到$B-1$这一范围内的散列表元号。

重点在于散列函数会“混杂”该元素。也就是说，h会混杂元素要落入的散列表元，这样一来这些元素大约就是会平均落入所有的散列表元中。即便元素本身相当有规律，比如是连续整数，或者只有一个位置不同的连续字符串，这种公平分配也一定会发生。

每个散列表元都是由链表组成的，该链表存储着散列函数发送给该散列表元的集合中的所有元素。要找到元素x，就要计算$h(x)$，得到散列表元号。如果x在，它肯定就在$h(x)$对应的散列表元中，这样我们可以沿着该散列表元对应的链表查找x。实际上，散列表让我们使用了较慢的集合的表实现，不过，通过将集合分为B个散列表元，让我们在查找表时平均只需要查找整个集合的$1/B$。如果让B差不多和集合的大小一样大，那么平均每个散列表元中就只有一个元素，这样查找元素平均只需要$O(1)$的时间了，就像在集合的特征向量表示中那样。

✦ 示例 7.15

假设我们要存储某字符串集合，每个字符串都以空字符结尾，而且最多只含32个字符。我们要使用上述第(2)条中提到的散列函数，其中B=5，也就是说，是有5个散列表元的散列表。要计算每个元素的散列值，就要求出每个字符串中直到空字符为止（但不包括空字符）各字符的整数值之和。以下定义给了我们想要的类型。

```
(1)     #define B 5
(2)     typedef char ETYPE[32];
(3)     DefCell(ETYPE, CELL, LIST);
(4)     typedef LIST HASHTABLE[B];
```

第(1)行定义了表示散列表元数量5的常量B。第(2)行定义的ETYPE类型是可容纳32个字符的数组。第(3)行是常见的链表及链表单元的定义，只不过这里的元素是ETYPE类型的，也就是32字符的数组。第(4)行将散列表定义为由B个链表组成的数组。如果接着定义

```
HASHTABLE headers;
```

headers数组有着包含散列表元头部的合适类型。

```
int h(ETYPE x)
{
    int i, sum;

    sum = 0;
    for (i = 0; x[i] != '\0'; i++)
        sum += x[i];
    return sum % B;
}
```

图7-11 假设ETYPE是字符数组，为与字符等价的整数求和的散列函数

现在必须定义散列函数h。该函数的代码如图7-11所示。与字符串x中各字符等价的整数会在变量sum中求和。最后一步会计算这个和除以散列表元数B得到的余数，并将其作为散列函数h的值返回。

下面拿一些单词作为例子，并考虑散列函数h安放这些单词的散列表元。要在散列表中输入7个单词[①]

```
anyone lived in a pretty how town
```

要计算h(anyone)，就需要搞清楚字符表示的整数值。在常用于表示字符的ASCII码中，小写字母对应的整数值从表示a的97（二进制的1100001）开始，到表示b的98，等等，直到表示z的122。而大写字母对应的整数要比相应小写字母对应的整数小32，也就是从表示A的65（二进制的1000001）到表示Z的90。

因此，与anyone中的字符对应的整数分别是97、110、121、111、110、101。它们的和是650。将这个和除以B，也就是5，得到余数为0。因此，anyone属于散列表元0。通过图7-11中的散列函数，就可以将本例中的7个单词分配到如图7-12所示的散列表元中。

单　词	和	散列表元
anyone	650	0
lived	532	2
in	215	0
a	97	2
pretty	680	0
how	334	4
town	456	1

图7-12　各个单词、它们的值和它们所在的散列表元

我们看到7个单词中有3个被分配到编号为0的散列表元中，有两个被分配到2号散列表元中，而1号和4号中各有一个单词。这与一般情况相比不那么平均，不过对少量的单词和散列表元来说，我们应该能预见这种不规则的情况。随着单词数变多，这些单词在5个散列表元中的分布就会近似平均了。插入了这7个单词之后的散列表如图7-13所示。

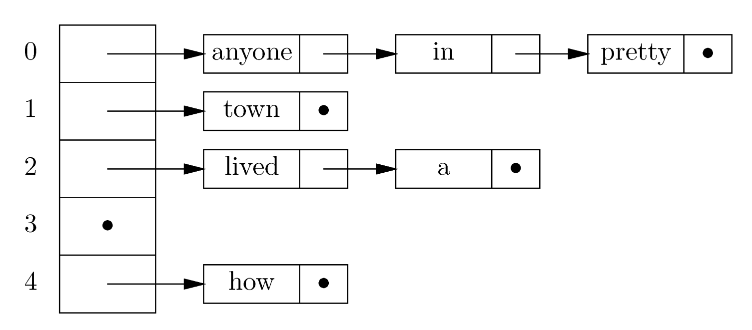


图7-13　存放7个元素的散列表

① 这些单词来自E.E.Cummings的一首同名诗，该诗的下一句是“with up so floating many bells down”。

7.6.2 词典操作的散列表实现

要在用散列表表示的词典中插入、删除或查找元素x，要经历简单的3步过程。

(1) 计算合适的散列表元，也就是$h(x)$。

(2) 利用由表头指针组成的数组，找到与标记为$h(x)$的散列表元对应的存储元素的表。

(3) 对该表执行操作，就像该表表示了整个集合一样。

针对这里的元素是字符串而6.4中的元素是整数这一事实，对6.4节中的算法经过恰当的修改之后，该算法可以用于这里的表操作。举例来讲，我们在图7-14中展示了向散列表插入元素的完整函数。大家可以自行开发`delete`和`lookup`函数作为练习。

```
     #include <string.h>

     void bucketInsert(ETYPE x, LIST *pL)
     {
(1)      if ((*pL) == NULL) {
(2)          (*pL) = (LIST) malloc(sizeof(struct CELL));
(3)          strcpy((*pL)->element, x);
(4)          (*pL)->next = NULL;
         }
(5)      else if (strcmp((*pL)->element, x)) /* x和element
                                                是不同的 */
(6)          bucketInsert(x, &((*pL)->next));
     }

     void insert(ETYPE x, HASHTABLE H)
     {
(7)      bucketInsert(x, &(H[h(x)]));
     }
```

图7-14 向散列表中插入元素

要理解图7-14，可以注意到函数`bucketInsert`与图6-5中的函数`insert`是相似的。在第(1)行，我们进行测试，看看是否已到达表的末端。如果是，就在第(2)行创建一个新单元。不过，在第(3)行，我们不再是把整数存储到新创建的单元中，而是利用标准头文件`string.h`里的`strcpy`函数将字符串x复制到该单元的元素字段。

还有，在第(5)行，我们会使用`string.h`中的`strcmp`函数测试是否尚未在该表中找到x。当且仅当x和当前单元的元素相等时，该函数会返回0。因此，只要这一比较的值非0，也就是只要当前元素不是x，我们就会沿着表继续向下。

这里的`insert`函数只有一行代码，在这行代码中，当我们找到对应适当散列表元$h(x)$头部的数组元素之后，就会调用`bucketInsert`。我们假设该散列函数h是在其他位置定义的。还要记得，类型`HASHTABLE`意味着`H`是指向各单元指针组成的数组（即链表数组）。

✦ 示例 7.16

假设我们要从图7-13所示的散列表中删除元素`in`，而使用的散列函数是示例7.15描述的。删除操作的执行方式从根本上讲与图7-14中的`insert`函数是类似的。我们会计算h(`in`)，其值为0。因此我们前往0号散列表元对应的表头。该散列表元对应的表中第二个单元存放着`in`，要删除该单元。具体的C语言程序留作本节习题。

7.6.3　散列表操作的运行时间

正如我们通过检视图7-14可以了解的，假设计算$h(x)$所花的时间是个与存储在散列表中的元素数量无关的常量，[①]函数`insert`找到适当散列表元头部所需的时间是$O(1)$。在这个常数的基础之上，还必须加上平均为$O(n/B)$的附加时间，其中n是散列表中的元素数量，而B则是散列表元的数量。原因在于，`bucketInsert`要花费与链表长度成比例的时间，而这一长度平均而言肯定是元素总数除以散列表元数，也就是n/B。

一个有趣的结果就是，如果让B约等于集合中元素的数量，也就是说，令n和B非常接近，则n/B大约为1，对散列表执行各种词典操作平均花费$O(1)$的时间，就和我们使用特征向量表示时一样了。如果尝试通过让B比n大得多来改善时间，会使多数散列表元为空，而这样做之后找到散列表元头部仍然要花$O(1)$的时间，因此让B比n大很多并不会显著改善运行时间。

还必须考虑到，在某些情况下，可能没法让B一直与n很接近。如果该集合增长迅速，那么n增加了而B仍然不变，最终n/B会变得很大。重组散列表是有可能的，只要通过为B选择一个更大的值，然后将每个元素都插入新的散列表中。完成这一工作需要$O(n)$的时间，不过这一时间不会大于向先前的散列表中插入n个元素所需的$O(n)$时间。请注意，这里的总时间$O(n)$是执行n次插入所花的时间，每次插入平均花费时间为$O(1)$。

7.6.4　习题

(1) 继续向图7-13中的散列表填充单词`with up so floating many bells down`。

(2) * 评价一下，下列散列函数在将常用英语单词集合分成大小基本相同的散列表元时，效率有多高。

  (a) 使用$B=10$，并设$h(x)$是单词长度x除以10得到的余数。

  (b) 使用$B=128$，并设$h(x)$是单词x最后一个字符的整数值。

  (c) 使用$B=10$。求单词x中各字符对应整数值的和。取求和结果的平方，然后取该结果除以10的余数。

(3) 使用与图7-14所示代码相同的假设，编写C语言程序，用于对散列表执行(a)删除；(b)查找操作。

7.7　关系和函数

尽管一般会假设集合中的元素都是原子的，不过在实践中让元素具有某种结构往往是很实用的。例如，在7.6节中我们谈论了长32个字符的字符串元素。另一种可作为元素的重要结构是定长表，它们和C语言的结构体类似。用作集合元素的表称为*元组*（tuple），表中每个元素称为元组的*组分*（component）。

元组中组分的数量称为元组的*元数*（arity）。例如，(a,b)是元数为2的元组，其第一个组分为a，第二个组分为b。元数为k的元组也称为k元组。

以具有相同元数（比方说是k）的元组为元素形成的集合称为*关系*。这一关系的元数就是k。元数为1的元组或关系是*一元的*。如果元数为2，就是*二元的*。一般来说，如果元数为k，那么元组或关系就是k*元的*。

① 这可能是图7-11所示散列函数的情况，也可能是实践中遇到的大多数散列函数的情况。计算散列表元编号的时间可能取决于元素的类型。例如，更长的字符串可能需要为更多的整数求和，但这一时间与存储的元素数量没关系。

✦ 示例 7.17

关系 $R=\{(1,2),(1,3),(2,2)\}$ 就是元数为2的关系，也就是二元关系。它的成员分别为(1,2)，(1,3)和(2,2)，都是元数为2的元组。

在本节中，我们主要考虑二元关系。还有很多非二元关系的重要应用，特别是在表列数据（就像在关系数据库中那样）的表示和操作中。我们将在第8章中进一步讨论该主题。

7.7.1 笛卡儿积

在正式研究二元关系之前，需要定义另一种集合运算。设A和B是两个集合，表示为$A\times B$的A和B的积，是指从A中选出第一个组分并从B中选出第二个组分所组成的有序对的集合，也就是

$$A\times B=\{(a,b)|a\in \text{A且}b\in \text{B}\}$$

该乘积有时也叫作*笛卡儿积*，是以法国数学家勒内・笛卡儿的名字命名的。

✦ 示例 7.18

回想一下，符号$\mathbf{Z}$约定俗成是表示所有整数的集合的。因此，$\mathbf{Z}\times\mathbf{Z}$就表示整数有序对的集合。

再举个例子，如果A是双元素集$\{1,2\}$，而B是三元素集$\{a,b,c\}$，那么$A\times B$就是6元素集$\{(1,a)$，$(1,b)$，$(1,c)$，$(2,a)$，$(2,b)$，$(2,c)\}$。

请注意，集合的积这一名称是名副其实的，因为如果A和B都是有限集，那么$A\times B$中元素的数量，正好是A中元素数量乘以B中元素数量的积。

7.7.2 两个以上集合的笛卡儿积

与算术积不同，笛卡儿积不具备交换律和结合律这些常规属性。很容易找出$A\times B\neq B\times A$的例子来推翻交换律。而结合律更是无从说起，$(A\times B)\times C$的成员有序对具有$((a,b),c)$的形式，而$A\times(B\times C)$的成员有序对则形如$(a,(b,c))$。

因为在很多时候需要谈论多元组的集合，所以需要将集合的积的表示法扩展到*k元笛卡儿积*。设$A_1\times A_2\times\cdots\times A_k$表示集合$A_1$、$A_2$、⋯、$A_k$的积，也就是说，满足$a_1\in A_1$且$a_2\in A_2$且⋯且$a_k\in A_k$的$k$元组$(a_1,a_2,\cdots,a_k)$的集合。

✦ 示例 7.19

$\mathbf{Z}\times\mathbf{Z}\times\mathbf{Z}$表示的是整数三元组$(i,j,k)$的集合，例如它包含了三元组(1,2,3)。不要把该三元笛卡儿积与表示有序对$((1,2),3)$的$(\mathbf{Z}\times\mathbf{Z})\times\mathbf{Z}$，或是表示有序对$(1,(2,3))$的$\mathbf{Z}\times(\mathbf{Z}\times\mathbf{Z})$弄混了。

另一方面，要注意到这3种乘积表达式都可以用由3个整数字段组成的结构体表示。不同之处在于解释结构体类型的方式。因此我们很容易混淆加括号和不加括号的乘积表达式。同样，以下三个C语言类型声明

```
struct {int f1; int f2; int f3;};
struct {struct {int f1; int f2;}; int f3;};
struct {int f1; struct {int f2; int f3;};};
```

都是以相似的方式存储的，只是存取字段的表示方式有所区别。

7.7.3 二元关系

二元关系R是作为集合A和集合B笛卡儿积子集的有序对集合。如果关系R是$A\times B$的子集，就说R是A到B的关系。而A就是该关系的定义域（domain），B就是该关系的值域（range）。如果B和A是相同集合，就说R是A上的关系，或者说是“定义域”A“上”的关系。

✦ 示例 7.20

整数上的算术关系$<$是$\mathbf{Z}\times\mathbf{Z}$的子集，由那些满足$a$小于$b$的有序对$(a,b)$组成。因此，符号$<$可被视作集合$\{(a,b)|(a,b)\in\mathbf{Z}\times\mathbf{Z}$，且$a$小于$b\}$的名称。然后我们用$a<b$作为“$(a,b)\in\ <$”或“$(a,b)$是关系$<$的成员”的简略形式。而整数上的其他算术关系，比如$>$或$\leqslant$，也可以按照相似的方式定义，而且实数上的算术比较都可以按照相似的方式定义。

再举个例子，考虑示例7.17中的关系R。它的定义域和值域是不确定的。我们知道1和2肯定在其定义域中，因为这两个整数是R中元组的第一个组分。同样，我们知道R的值域肯定包含2和3。不过，可将R看作是$\{1,2\}$到$\{2,3\}$的关系，或是将其视作$\mathbf{Z}$到$\mathbf{Z}$的关系，这只是无数选择中的两个例子而已。

7.7.4 关系的中缀表示

正如我们在示例7.20中所表示的，二元关系的中缀表示法是很常用的，所以，像$<$关系这样本来是有序对的集合，却可以写在关系中各有序对的两个组分之间。这也就是为什么我们通常会看到诸如$1<2$和$4\geqslant4$这样的表达式，而不是看到更为学究式的$(1,2)\in\ <$或$(4,4)\in\ \geqslant$。

✦ 示例 7.21

关系的中缀表示法可以用于任意类型的二元关系。例如，示例7.17中的关系R就可以写为3个“事实”$1R2$、$1R3$和$2R2$。

声明的及当前的定义域和值域

示例7.20的第二部分强调了一点，就是不能只从看到的表象来断定关系的定义域和值域。作为第一个组分出现的元素组成的集合肯定是定义域的子集，而作为第二个组分的元素组成的集合一定是值域的子集。不过，在定义域或值域中还可能有其他的元素。

当关系不发生改变时，这种差异是不重要的。不过，我们在7.8节和7.9节，以及在第8章的内容中会看到，值会发生改变的关系是非常重要的。例如，我们可能谈论某一关系，其定义域是某门课程中的学生，而值域则是一些整数，表示作业的总分。在开课之前，该关系中是没有有序对的。在第一次作业被评分后，每个学生就各有了一个有序对。随着时间的推移，会有学生弃选这门课程，或是有学生加入该课程，而总分在不断增加。

我们可以将该关系的定义域定义为所有在该大学注册的学生，而将值域定义为整数的集合。当然，不论何时，该关系的值都是这两个集合笛卡儿积的子集。另一方面，不管什么时候，关系都具有当前定义域和当前值域，就是由出现在关系中有序对第一个组分和第二个组分位置的元素分别构成的集合。当我们需要加以区分时，就会将关系本来的定义域和值域称作声明的定义域和值域。当前的定义域和值域分别是声明的定义域和值域的子集。

7.7.5 表示二元关系的图

可以用图来表示定义域为A且值域为B的关系R。先为在A和（或）B中的每个元素画一个节点。如果aRb，就画一条从a到b的箭头（“弧”），我们将在第9章中更详尽地讨论一般图。

✦ 示例 7.22

表示示例7.17中关系R的图如图7-15所示。它有表示1、2、33个元素的3个节点。因为$1R2$，所以从节点1到节点2有一条弧。因为$1R3$，所以有一条从1到3的弧。而且有$2R2$，所以有一条从节点2到它本身的弧。除此之外没有其他的弧，因为R中不再包含其他有序对了。

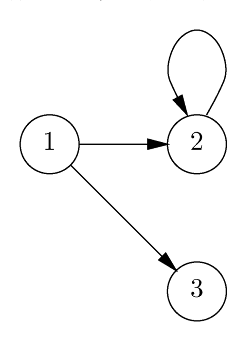


图7-15 表示关系 $\{(1,2),(1,3),(2,2)\}$ 的图

7.7.6 函数

假设有从定义域A到值域B的关系R具有如此属性：对其定义域A中个每个成员a而言，在其值域B中最多有一个b满足aRb。这样的R就被称作从定义域A到值域B的偏函数。

如果对A中每个成员a来说，都刚好在B中有一个元素b满足aRb，就说R是从A到B的全函数。偏函数和全函数之间的区别在于，偏函数可能对其定义域中的某些元素而言无定义，例如，对A中的某个a，可能在B中不存在满足aRb的b。我们会使用术语“函数”来指代偏函数更为一般化的概念，不过，只要偏函数与全函数之间的区别关系重大，我们就会用上“偏”字。

有一种常用的函数表示法，如果b是满足aRb的唯一元素，通常就写成 $R(a)=b$ 。

✦ 示例 7.23

设S是由 $\{(a,b)\,|\,b=a^2\}$（也就是第二个组分为第一个组分平方的有序对的集合）给出的从$\mathbf{Z}$到$\mathbf{Z}$的全函数。那么S具有诸如(3,9)、(–4,16)和(0,0)这样的成员。我们可以通过写出 $S(3)=9$ 、$S(-4)=16$ 和 $S(0)=0$ 来表示S为平方函数这一事实。

请注意，函数在集合论中的概念与C语言中函数的概念没有太大区别。也就是说，假设s是具有如下声明

```
int s(int a)
{
    return a*a;
}
```

的C语言函数，它接受一个整数作为参数并返回该整数的平方。我们通常会将`s(a)`视为与$S(a)$相同的函数，尽管前者是计算平方的一种方式，而后者只是抽象地定义了求平方的运算。还要

注意到，在实际应用中，`s(a)`总是偏函数，因为出于计算机算术能力的限制，有很多`a`的值让`s(a)`不会返回整数。

C语言中也有接受多个参数的函数。接受两个整数参数`a`和`b`，并返回一个整数的C语言函数`f`，就是从$\mathbf{Z}\times\mathbf{Z}$到**Z**的函数。同样，如果两个参数分别有着让它们分属集合A和集合B的类型，而`f`返回的是类型C的某个成员，那么`f`就是从$A\times B$到C的函数。更一般地讲，如果函数`f`接受分别来自集合A_1、A_2、…、A_k的k个参数，并返回集合B的某个成员，我们就说`f`是从$A_1\times A_2\times\cdots\times A_k$到$B$的函数。

例如，可以将6.4节中的`lookup(x,L)`函数视作从$\mathbf{Z}\times L$到{`TRIE`，`FALSE`}的函数。这里的L是整数链表的集合。

函数的多种表示法

从$A\times B$到C的函数F从理论上讲是$(A\times B)\times C$的子集。因此函数F中的有序对都应该具有$\big((a,b),c\big)$这样的形式，其中a、b、c分别是集合A、B、C的成员。使用函数的特别表示法，可以写成$F(a,b)=c$。

还可将F视作从$A\times B$到C的关系，因为每个函数都是一个关系。使用关系的中缀表示法，$\big((a,b),c\big)$在F中这一事实也可以写为$(a,b)Fc$。

在将笛卡儿积扩展到多个集合时，我们可能希望从乘积表达式中删除括号。因此，我们可能将$(A\times B)\times C$视为技术上讲与其不相等的表达式$A\times B\times C$。在这种情况下，F的成员就可以写为(a,b,c)。如果将F存储为这种三元组的集合，就一定要记住前两个组分一起组成定义域元素，而第三个组分是值域元素。

正式地讲，从定义域$A_1\times A_2\times\cdots\times A_k$到值域$B$的函数，就是形如$\big((a_1,\cdots,a_k),b\big)$的有序对的集合，其中$a_i$是集合$A_i$的成员，$b$是集合$B$的成员。请注意，该有序对的第一个元素本身也是个$k$元组。例如，上面提到的`lookup(x,L)`函数也可以视作有序对$\big((x,L),t\big)$的集合，其中$x$是整数，$L$是整数链表，而$t$要么是`TRUE`要么是`FALSE`，具体取决于$x$是否在链表$L$中。不管函数是用C语言编写的，还是在集合论中正式定义的，都可以将其视为一个从定义域集合接受某个值并生成值域中某个值的容器，如表示函数`lookup`的图7-16所示。

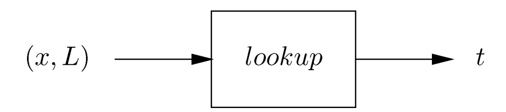


图7-16　函数将定义域中的元素与值域中唯一的元素关联起来

7.7.7　一一对应

设从定义域A到值域B的偏函数F具有下列属性。

(1) 对A中的每个元素a，在B中都有一个元素b满足$F(a)=b$。

(2) 对B中的每个b，在A中都存在某个a满足$F(a)=b$。

(3) 在B中没有这样的b，使得A中有两个元素a_1和a_2满足$F(a_1)$和$F(a_2)$都是b。

这样的F就称为从A到B的一一对应。而这种一一对应也可以用术语双射（bijection）来表示。

属性(1)表示F是从A到B的全函数。属性(2)是表示F是从A到B之上的全函数的条件。一些数学家会使用术语满射（surjection）来表示这种从A到B之上的全函数。

属性(2)和属性(3)一起表示F就像从B到A的全函数那样。而具有属性(3)的全函数有时也被称为单射（injection）。

一一对应基本上就是两个方向上的全函数，不过要注意到，F是否为一一对应不止取决于F中的有序对，还取决于声明的定义域和值域。例如，可以取任意从A到B的一一对应，并通过向A中增加某个在F中未提及的新元素e而改变定义域。这样F就不会是从 $A\cup\{e\}$ 到B的一一对应。

✦ 示例 7.24

示例7.23中从$\mathbf{Z}$到$\mathbf{Z}$的求平方函数S就不是一一对应。它确实满足属性(1)，因为对每个整数i，都存在某个整数，也就是i^2，满足 $S(i)=i^2$ 。不过，它不满足属性(2)，因为对某些在$\mathbf{Z}$中的b，具体来说就是所有的负整数，在$\mathbf{Z}$中不存在a使得 $S(a)=b$ 。S也不满足属性(3)，因为存在很多两个不同的a使 $S(a)$ 等于同一个b的例子。例如， $S(3)=9$ ，而且 $S(-3)=9$ 。

要举一一对应的例子，可以考虑定义为 $P(a)=a+1$ 的从$\mathbf{Z}$到$\mathbf{Z}$的全函数P。也就是说，P会为任一整数加1。例如， $P(5)=6$ ，而且 $P(-5)=-4$ 。还可以将P视作由二元组形成的集合 $\{\cdots,(-2,-1),(-1,0),(0,1),(1,2),\cdots\}$ ，或者是图7-17所示的图。

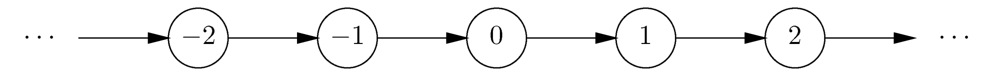


图7-17 表示函数 $P(a)=a+1$ 这一关系的图

我们声明P是从整数到整数的一一对应。首先，这是个偏函数，因为当为整数a加上1时，可得到唯一的整数 $a+1$ 。它是满足属性(1)的，因为对每个整数a，存在某个作为 $P(a)$ 的整数 $a+1$ 。属性(2)也得到满足，因为对每个整数b，都存在某个整数，即 $b-1$ ，满足 $P(b-1)=b$ 。最后，属性(3)也是满足的，因为对某个整数b而言，不存在这样两个不同的整数，使得给这两个整数各自加上1后都得到b。

从A到B的一一对应是在A和B的元素之间构建唯一关联的一种方式。例如，如果双手合十，左手和右手的大拇指触在一起，左手和右手的食指触在一起，等等。我们可以把左手手指集合与右手手指集合之间的这种关联看作一一对应F，定义为F(“左拇指”) =□右拇指□，F(“左食指”) =□右食指□，等等。也可以将这种关联看作F的逆函数，也就是从右手到左手的函数。总的说来，可以通过调换有序对中组分的次序，反转从A到B的一一对应，从而成为从B到A的一一对应。

左右手之间存在这种一一对应的结果就是每只手手指的数量是相同的。这似乎是种自然而且直觉上的概念，当一个集合到另一个集合正好存在一一对应时，这两个集合有着相同数量的元素。不过，我们在7.11节中会看到，当集合为无限集时，从这一“元素数量相同”的定义会得出一些惊人的结论。

7.7.8 习题

(1) 给出使$A\times B$不同于$B\times A$的集合A和B的例子。

(2) 设R是由aRb、bRc、cRd、aRc和bRd定义的关系。

(a) 画出表示R的图。

(b) R是否为函数？

(c) 为R指出两个可能的定义域，并指出两个可能的值域。

(d) 满足R是S上关系（即定义域和值域都可以是S）的最小集合S是什么？

(3) 设T是树，并设S是树T的节点的集合。设R是节点间的“父子”关系，也就是说，当且仅当c是p的子节点时有cRp。回答以下问题，并验证子集的答案。

(a) 不管树T是什么，R是否为偏函数？

(b) 不管树T是什么，R是否为从S到S的全函数？

(c) R有没有可能是一一对应（即对某树T而言）？

(d) 表示R的图是什么样的？

(4) 设R是整数集合$\{1,2,\cdots,10\}$上的关系，其中如果a和b是不同整数而且有除1之外的公约数，就说aRb。例如，$2R4$，$6R9$，但是没有$2R3$。

(a) 画出表示R的图。

(b) R是否为函数？为什么？

(5) * 虽然我们看到$S=(A\times B)\times C$和$T=A\times(B\times C)$是不同的集合，但是通过展示出它们之间存在的自然的一一对应，可以证明它们“从根本上讲是相同的”。对S中的每个$\big((a,b),c\big)$而言，设$F\Big(\big((a,b),c\big)\Big)=\big(a,(b,c)\big)$。证明$F$是从$S$到$T$的一一对应。

(6) $F(10)=20$、$10F20$和$(10,20)\in F$这3项陈述有何共同之处。

(7) *关系R的逆关系（简称R的逆）是指满足(a,b)在R中的有序对(b,a)的集合。

(a) 说明如何从表示R的图得出表示R的逆的图。

(b) 如果R是全函数，那么R的逆是否一定为函数？如果R是一一对应呢？

(8) 证明：当且仅当某关系及其逆关系都是全函数时，该关系是一一对应。

7.8　将函数作为数据来实现

在程序设计语言中，函数通常是由代码实现的，不过当它们的定义域很小时，可以使用相当类似于实现集合的技巧来实现它们。我们在本节中要讨论如何使用链表、特征向量和散列表来实现有限函数。

作为程序的函数与作为数据的函数

尽管7.7节中我们在函数的抽象概念与C语言中实现的函数间作了很强的类比，不过还是应该注意到它们间的重大差别。如果F是C语言函数，而x是其定义域集合中的成员，那么F就告诉了我们如何计算$F(x)$的值。而同样的程序对任意的值x都是起作用的。

然而，当我们将函数表示为数据时，首先就需要函数是由有序对的有限集构成。其次，通常这些有序对基本是不可预测的。也就是说，在给定x的情况下，没什么方便的办法来计算$F(x)$的值。我们能做的最佳做法就是创建表给出每个满足$F(a_i)=b_i$的有序对

$$(a_1, b_1),\ (a_2, b_2),\ \cdots,\ (a_n, b_n)$$

这样的函数事实上是数据，而不是程序，尽管原则上讲可以创建程序，将这样的表存储为该程序的一部分，并在给定x的情况下从内部表中查找$F(x)$。不过，更加高效的做法是将该表单独储存为数据，并利用可以处理任一这种函数的通用算法来进行值的查找。

7.8.1　对函数的操作

最常对函数执行的操作与对词典的操作类似。假设F是从定义域集合A到值域集合B的函数。那么我们可以进行下述操作

(1) 插入满足 $F(a)=b$ 的新有序对 (a,b) 。唯一的微小区别在于，因为F一定是函数，所以假如其中已经存在有序对 (a,c) ，那么该有序对肯定会被 (a,b) 替代。

(2) 删除与 $F(a)$ 关联的值。在这里，我们只需要给出定义域中的值a即可。如果存在b满足 $F(a)=b$ ，有序对 (a,b) 就会从集合中删除。如果没有这样的有序对，就不会发生任何改变。

(3)查找与 $F(a)$ 关联的值，也就是说，给定定义域中的值a，返回满足 $F(a)=b$ 的值b。如果集合中没有这样的有序对 (a,b) ，就返回某个特殊的值来警告 $F(a)$ 是未定义的。

✦　示例 7.25

假设F由有序对 $\{(3,9),(-4,16),(0,0)\}$ 组成，也就是说，$F(3)=9$ 、$F(-4)=16$ 而且 $F(0)=0$ 。那么*lookup*(3)会返回9，而*lookup*(2)则返回一个特殊的值，指示没有值被定义为 $F(2)$ 。如果F是“求平方”函数，那么值−1可以用来指示不存在的值，因为−1不可能是任何整数真正的平方值。

操作*delete*(3)会删除有序对(3,9)，*delete* (2)则没有效果。如果执行*insert*(5,25)，那么会在集合F中添加有序对(5,25)，或者说现在有了 $F(5)=25$ 。如果执行*insert*(3,10)，就会从F中删除旧的有序对(3,9)，并将新的有序对(3,10)添加到F中，这样一来就有了 $F(3)=10$ 。

7.8.2　函数的链表表示

函数作为有序对集合，可以像其他任何集合那样存储在链表中。定义含有3个字段的单元是很实用的，一个表示定义域的值，另一个表示值域的值，最后一个表示指向下一个单元的指针。例如，我们可以按照如下方式定义单元。

```
typedef struct CELL *LIST;
struct CELL {
    DTYPE domain;
    RTYPE range;
    LIST next;
};
```

其中`DTYPE`是定义域元素的类型，而且`RTYPE`表示值域元素的类型。那么函数就可以表示为指向链表（第一个单元）的指针。

图7-18中的函数执行操作*insert*(a,b,L)，假设`DTYPE`和`RTYPE`都是32字符的数组。我们查找在`domain`字段中含有值a的单元。如果找到，就将其`range`字段置为b。如果到达链表末端，就创建一个新单元，并将 (a,b) 存储进去。否则，测试该单元中是否含有定义域元素a。如果有，那么就将值域的值改为b，这样就行了。如果定义域中有a之外的值，就递归地将其插入链表尾部。

如果函数F中有n个有序对，那么插入操作平均要花 $O(n)$ 的时间。同样，用于表示为链表的函数的`delete`和`lookup`函数也平均需要 $O(n)$ 的时间。

```
typedef char DTYPE[32], RTYPE[32];

void insert(DTYPE a, RTYPE b, LIST *pL)
{
    if ((*pL) == NULL) {/* 在表的末端 */
        (*pL) = (LIST) malloc(sizeof(struct CELL));
        strcpy((*pL)->domain, a);
        strcpy((*pL)->range, b);
        (*pL)->next = NULL;
    }
    else if (!strcmp(a, (*pL)->domain)) /* domain 字段是 a,
                                           改变 F(a) */
        strcpy((*pL)->range, b);
    else /* domain 字段不是 a */
        insert(a, b, &((*pL)->next));
};
```

图7-18 将新事实插入表示为链表的函数中

7.8.3 函数的向量表示

假设声明的定义域是从0到*DNUM*-1的整数，也可以通过枚举类型来定义。然后我们可以使用特征向量的表示函数，将表示特征向量的类型FUNCT定义为：

```
typedef RTYPE FUNCT[DNUM];
```

这是判断该函数为全函数或者RTYPE包含一个可以解释为“值不存在”的值的关键所在。

✦ 示例 7.26

假设我们要存储与苹果有关的信息，就像图7-9中的收获期信息，不过现在希望给出具体的收获月份，而不是早熟/晚熟这种二元的选择。通过定义如下枚举类型，我们为定义域和值域中的每个元素都关联了一个整数常量：

```
enum APPLES {Delicious, GrannySmith, Jonathan, McIntosh,
             Gravenstein, Pippin};
enum MONTHS {Unknown, Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,
             Sep, Oct, Nov, Dec};
```

这一声明将0与标识符Delicious关联，将1与GrannySmith关联，等等。它还将0与Unknown关联，将1与Jan关联，等等。标识符Unknown表示收获月份是未知的。现在可以声明数组

```
int Harvest[6];
```

用该Harvest数组表示图7-19所示的有序对集合。接着数组Harvest就成了图7-20那样，其中数据项Harvest[Delicious] = Oct意味着Harvest[0] = 10。

苹　　果	收获月份
美味（Delicious）	十月（Oct）
格兰尼·史密斯（Granny Smith）	八月（Aug）
乔纳森（Jonathan）	九月（Sep）
旭苹果（McIntosh）	十月（Oct）
格拉文施泰因（Gravenstein）	九月（Sep）
翠玉苹果（Pippin）	十一月（Nov）

图7-19 苹果的收获月份

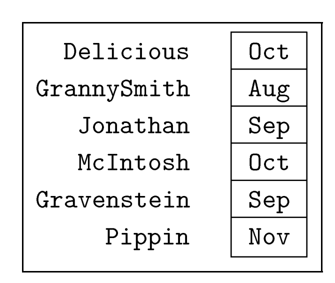


图7-20 Harvest数组

7.8.4 函数的散列表表示

我们可以将属于某函数的有序对存储在散列表中。关键的是，我们只对定义域的元素应用散列函数，以确定有序对所属的散列表元。形成散列表元的链表单元都有一个表示定义域元素的字段，而另一字段表示对应的值域元素，第三个字段则是将链表中的一个单元链接到下一个单元。下面举个例子，应该就能把这个技巧说清了。

✦ 示例 7.27

我们继续使用示例7.26中有关苹果的数据，不过现在要使用实际名称来表示定义域。要表示函数Harvest，我们会使用含5个散列表元的散列表。这里要将APPLES定义为32字符的数组，而MONTHS还是示例7.26中那样的枚举。散列表元是链表，具有表示APPLES类型定义域元素的variety字段、表示int类型（月份）值域元素的harvested字段，以及指向链表中下个元素的链接字段next。

我们会使用与7.6节中图7-11类似的散列函数h。当然，h只会应用到定义域元素上，也就是说，只会应用到由苹果品种名组成的长32个字符的字符串上。

现在，可以将类型HASHTABLE定义为B个LIST组成的数组。其中B是散列表元的数量，我们已经将其定为5了。所有这些声明都出现在图7-22的开头。然后就可以声明散列表Harvest来表示所需的函数。

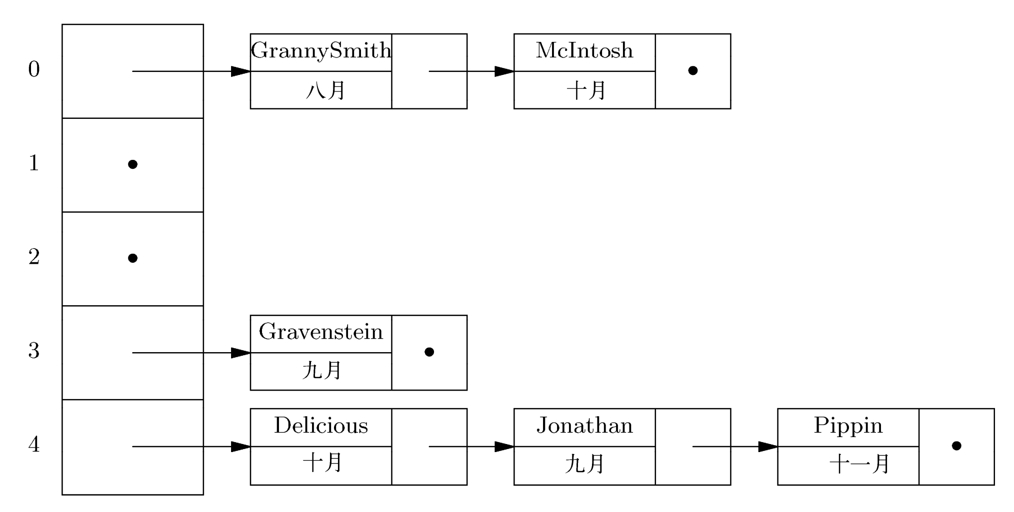


图7-21 存储在散列表中的苹果品种名称及其收获月份

在插入图7-19中列出的6个苹果品种之后，散列表元中单元的分布如图7-21所示。例如，如果将单词`Delicious`的9个字符对应的整数值加起来，就得到929。因为929除以5的余数为4，所以美味苹果（Delicious）就属于4号散列表元。而表示该苹果品种的单元将字符串`Delicious`存放在`variety`字段中，将月份`Oct`存放在`harvested`字段中，最后还有一个指向散列表元中下一个单元的指针。

```
#include <string.h>
#define B 5

typedef char APPLES[32];
enum MONTHS {Unknown, Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,
          Sep, Oct, Nov, Dec};
typedef struct CELL *LIST;
struct CELL {
    APPLES variety;
    int harvested;
    LIST next;
};
typedef LIST HASHTABLE[B];

int lookupBucket(APPLES a, LIST L)
{
    if (L == NULL)
        return Unknown;
    if (!strcmp(a, L->variety)) /* 找到 */
        return L->harvested;
    else /* 未找到a，检查尾部 */
        return lookupBucket(a, L->next);
}

int lookup(APPLES a, HASHTABLE H)
{
    return lookupBucket(a, H[h(a)]);
}
```

图7-22　用于通过散列表表示的函数的查找

7.8.5　对用散列表表示的函数的操作

要执行插入、删除和查找操作，都要从需要散列的定义域值从而找到散列表元开始。要插入有序对(a,b)，就要找到散列表元$h(a)$，并查找它对应的链表。接下来的操作就和图7-18中给出的向链表插入函数有序对的函数一样了。

要执行`delete(a)`，先要找到散列表元$h(a)$，查找具有定义域值a的单元，要是找到这样的单元，就从链表中删除该单元。而执行$lookup(a)$操作还是要散列a，然后在散列表元$h(a)$中查找含有定义域值a的单元。如果找到这样的单元，就会返回与之对应的值域值。

例如如图7-22所示的函数`lookup(a,H)`。函数`lookupBucket(a,L)`会沿着与某散列表元对应的链表L向下查找，并返回值`harvested(a)`，也就是苹果品种a收获的月份。如果这一月份是未定义的，就返回值`Unknown`。

向量和散列表

示例7.26和示例7.27中看待有关苹果的信息的方式有着根本的区别。在特征向量法中，苹果品种是个固定集，是枚举类型的。当C语言程序正在运行时，是没办法改变苹果名称集合的，而且对一个未出现在枚举集合中的名称执行查找也是没意义的。

另一方面，当我们用散列表来构建同一函数时，是将苹果名称作为字符串，而不是枚举类型的数字。这样一来，就有可能在程序正在运行时对名称集合进行修改了，比方说是为了响应某些与新的苹果品种有关的输入数据。对散列表中未出现的品种执行查找是可行的，而且我们必须有所防备，要加上`Unknown`这样一个“月份”，以防出现查找散列表中未提及品种的情况。因此，散列表的灵活性要比特征向量更佳，不过要付出一些速度上的代价。

7.8.6　函数操作的效率

对以我们在本节中讨论过的这3种方式表示的函数执行各种操作所需的时间，与对词典执行同样操作所需的时间是一样的。也就是说，如果函数由n个有序对组成，那么链表表示下每种操作平均需要$O(n)$的时间。特征向量法每种操作只需要$O(1)$的时间，不过，就像词典那样，只有定义域类型的大小比较有限时，才能使用该表示法。而具有B个散列表元的散列表每种操作的平均时间是$O(n/B)$。如果有可能让B接近n，那么就可以达到每种操作平均花费$O(1)$时间的水平。

7.8.7　习题

(1) 模仿图7-18中的`insert`函数，编写函数，对用链表表示的函数执行(a)删除；(b)查找操作。

(2) 编写函数，对用向量表示的函数，也就是由`DTYPE`类型的整数作为下标的`RTYPE`类型的数组，执行(a)插入；(b)删除和(c)查找操作。

(3) 模仿图7-22中的`lookup`函数，编写函数，对用散列表表示的函数执行(a)插入；(b)查找操作。

(4) 二叉查找树也可用来表示作为数据的函数。为二叉查找树定义合适的数据结构，以存放图7-19中的苹果信息，并使用这些数据结构实现(a)插入；(b)删除；(c)查找操作。

(5) 设计一个信息检索系统，记录有关棒球球员击球和击中的信息。所设计的系统应该接受形如`Ruth 5 2`的三元组，表示Ruth在5次击球中击中了2次。对应Ruth的数据项应该得到适当的更新。大家应该还能查询任意球员的击球次数和击中次数。实现该系统，使得只要执行插入和查找操作的函数使用了合适的子程序和类型，就对任意数据结构都有效。

7.9　二元关系的实现

二元关系的实现与函数的实现有些许差异。回想一下，二元关系与函数都是有序对的集合，不过在函数中，对定义域中的各元素a来说，最多只能与任一值域元素b构成一个形如(a,b)的有序对。而二元关系则不同，可以有任意数量的值域元素与某个给定的定义域元素a相关联。

在本节中，我们首先会考虑二元关系的插入、删除和查找操作的意义。然后看看已经用到的3种实现——链表、特征向量和散列表——是如何一般化到二元关系上的。在第8章中，我们会讨论多元关系的实现。通常，表示多元关系的数据结构，都是构建在表示函数和二元关系的数据结构的基础之上的。

7.9.1　对二元关系的操作

当我们将有序对 (a,b) 插入二元关系R中时，并不需要关心R中是否已经存在某个有序对 (a,b)，其中 $c \neq b$，但向某个函数中插入 (a,b) 时，就需要关心这个了。原因当然是R中包含定义域值a的有序对数量是没有限制的。因此，可以直接将有序对 (a,b) 插入R中，就像将元素插入任意集合中那样。

同样，从关系中删除有序对 (a,b) 也类似于从集合中删除元素：要查找该有序对，如果存在就将其删除。

查找操作可以用多种方式定义。例如，我们可以接受有序对 (a,b)，并询问该有序对是否在R中。不过，如果我们因此将对关系的查找操作解释成与刚定义的插入和删除操作那样，与对任意词典的这些操作行为相同，被操作的元素是有序对而不是原子这一事实就只是个小细节，它只能影响到词典中元素的类型。

然而，定义`lookup`来接受定义域元素a，并返回所有满足 (a,b) 在二元关系R中的值域元素b往往是很实用的。对`lookup`的这种解释给了我们一种与词典有所区别的抽象数据类型，它有着与词典ADT不同的某些特定用途。

✦　示例 7.28

大多数李子品种需要另一种特定的品种来传粉，没有合适的“传粉者”，这棵李树就不会结果。有少数品种是“自育的”，也就是说它们可以作为自己的传粉者。图7-23展示了李子品种集合上的二元关系。这一关系中的有序对 (a,b) 表明品种b是品种a的传粉者。

将有序对插入该表表示断言某个品种是另一个品种的传粉者。例如，如果培育出新品种，就可能要向该关系中输入与可以给该新品种传粉的品种以及可以被它传粉的品种有关的事实。删除某个有序对，就表示收回某个品种可为另一品种传粉的断言。

品　　种	传粉者
美丽（Beauty）	圣罗莎（Santa Rosa）
圣罗莎（Santa Rosa）	圣罗莎（Santa Rosa）
伯班克（Burbank）	美丽（Beauty）
伯班克（Burbank）	圣罗莎（Santa Rosa）
澳得罗达（Eldorado）	圣罗莎（Santa Rosa）
澳得罗达（Eldorado）	威克森（Wickson）
威克森（Wickson）	圣罗莎（Santa Rosa）
威克森（Wickson）	美丽（Beauty）

图7-23　某些李子品种的传粉者

对关系更一般的操作

除了对示例7.28中的李子品种进行插入、删除和查找这3种操作可以提供的信息之外，我们可能还需要更多的信息。例如，我们可能想问，“圣罗莎可以为哪些品种传粉？”或者“澳得罗达能否给美丽传粉？”某些数据结构，比如链表，让我们能以执行这3种基本词典操作的速度回答这样的问题，只要不是链表对这些操作很低效。

基于定义域元素的散列表无助于回答给定了值域元素并必须找到对应定义域元素的问题，例如，“圣罗莎可以为哪些品种传粉？”当然，可以对值域元素应用散列函数，不过这样一来就不好回答“什么品种可以给伯班克传粉？”这样的问题了。还可以对定义域元素和值域元素的组合应用散列函数，不过这样一来对哪种类型的查询都不能高效响应了，只能回答一些类似“澳得罗达能否给美丽传粉？”这样的简单问题。

有多种方式能高效地回答所有这些类型的问题。不过，我们要等到第8章谈论关系模型时才会了解到这些技巧。

我们定义的查找操作接受变量a作为参数，查看第一列，寻找所有包含值a的有序对，并返回与之关联的值域值集合。也就是说，询问“哪个品种可以给品种a传粉？”该问题似乎是最可能询问的与该表有关的信息，因为如果我们种植了一棵李树，就必须确认，如果它不是自育的，就应该在附近种植传粉者。例如，如果调用`lookup(Burbank)`，预期答案就是{Beauty，Santa Rosa}。

7.9.2 二元关系的链表实现

如果愿意的话，我们可以将关系中的有序对在链表中链接起来。该链表的单元都含有一个定义域元素、一个值域元素，以及一个指向下一个单元的指针，就像表示函数的链表单元那样。插入和删除操作，就像6.4节中讨论过的针对一般集合的插入和删除那样。唯一的小差别就是这里集合成员的相等性，是通过比较存放定义域元素的字段以及存放值域元素的字段确定的。

这里的查找操作要与我们之前遇到的查找操作有些不同。我们必须沿着链表向下，查找含某个特定定义域值a的单元，而且必须将与之相关的值域值组成一个链表。下面的示例将会展示对链表进行查找操作的机制。

✦ 示例 7.29

假设我们想用链表来实现示例7.28中的李子关系。可以将`RVARIETY`类型定义为长32个字符的字符串，并将类型为`RCELL`（relation cell，关系单元）的单元定义为结构体

```
typedef char PVARIETY[32];
typedef struct RCELL *RLIST;
struct RCELL {
    PVARIETY variety;
    PVARIETY pollinizer;
    RLIST next;
};
```

我们还需要一个单元容纳一个李子品种和指向下一个单元的指针，以构建某给定品种传粉者的链表，并以此来回应`lookup`查询。我们将该类型称为`PCELL`，并定义

```
typedef struct PCELL *PLIST;
struct PCELL {
    PVARIETY pollinizer;
    PLIST next;
};
```

然后可以通过图7-24中的函数定义查找操作。

函数`lookup`接受定义域元素a和指向有序对链表第一个单元的指针作为参数。通过调用`lookup(a,L)`，可以对关系R执行`lookup(a)`操作，这里的L是指向表示关系R的链表第一个单元的指针。第(1)行和第(2)行都很简单。如果链表为空，就返回`NULL`，因为在空链表中不存在第一个组分为a的有序对。

```
     PLIST lookup(PVARIETY a, RLIST L)
     {
         PLIST P;

(1)      if (L == NULL)
(2)          return NULL;
(3)      else if (!strcmp(L->variety, a)) /* L->variety == a */ {
(4)          P = (PLIST) malloc(sizeof(struct PCELL));
(5)          strcpy(P->pollinizer, L->pollinizer);
(6)          P->next = lookup(a, L->next);
(7)          return P;
         }
         else /* a 不是当前数对的定义域值 */
(8)          return lookup(a, L->next);
     }
```

图7-24　在用链表表示的二元关系中进行查找

难题就是在链表第一个单元的定义域字段`variety`中找到a的情况。这种情况是在第(3)行检测，在第(4)行至第(7)行得到处理的。我们在第(4)行创建一个`PCELL`类型的新单元，这将成为我们要返回的`PCELL`链表中的第一个单元。第(5)行会将相关联的值域值复制到新单元中。然后在第(6)行我们会对链表L的尾部递归地调用`lookup`。该调用的返回值是指向得到的链表中第一个单元的指针（如果链表为空则是`NULL`），它会成为我们在第(4)行中所创建单元的`next`字段。然后第(7)行要返回指向新创建单元的指针，该单元存放着对应定义域值a的一个值域值，而且如果存在对应a的其他值域值，该单元还将链接到存放其他值域值的单元。

最后一种情况是没有在链表L的第一个单元中找到所需的定义域值a。这时只要在第(8)行对链表L的尾部调用`lookup`，并返回该调用返回的任何内容即可。

7.9.3　特征向量法

我们看到，对于集合与函数，可以通过创建以某个“全集”的元素为索引的数组，并在数组中放置合适的值来表示这些集合与函数。对集合来说，合适的数组值就是`TRUE`和`FALSE`，而对函数而言，就是那些可以出现在值域中的值，通常还要加上表示“无”的特殊值。

对二元关系来说，可以通过某个较小的声明定义域中的成员作为数组的索引，就像处理函数时那样。不过，不能使用单个值作为数组元素，因为在二元关系中，对于某个给定的定义域值，可能有任意数量的值域值与之对应。最好是把与某给定定义域值相关联的所有值域值存入一个链表，然后将该链表的表头作为数组的元素。

✦　示例 7.30

我们用这种组织方式再来处理李子品种的例子。正如我们在7.8节中指出的，在使用特征向量表示法时，必须让值的集合固定不变，至少要保证定义域值的集合不变，而对链表或散列表的表示而言，就不存在这种限定。因此，必须重新将`PVARIETY`类型声明为枚举类型

```
enum PVARIETY {Beauty, SantaRosa, Burbank, Eldorado, Wickson};
```

我们可以继续使用示例7.29中定义的表示品种链表的`PCELL`类型，这样就可以将数组定义为

```
PLIST Pollinizers[5];
```

也就是说，表示图7-23所示关系的数组，是用该图中提及的品种作为索引的，而与每个品种关

联的值，都是指向其传粉者链表第一个单元的指针。图7-25展示了用特征向量法表示出的图7-23中的有序对。

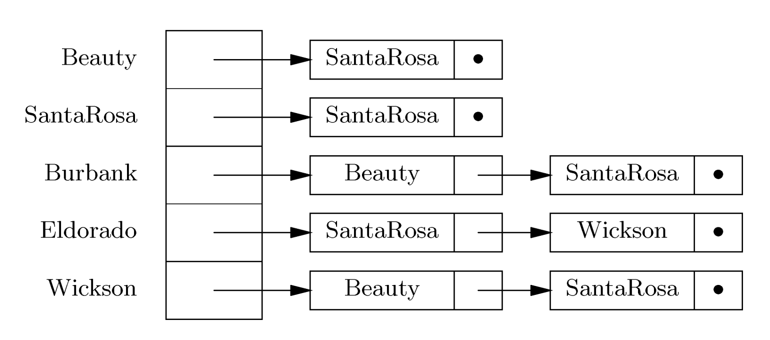


图7-25　传粉者关系的特征向量表示

要执行有序对的插入和删除，先要找到恰当的数组元素，并从那里开始沿着链表行进。至此，链表的插入和删除操作就很平常了。例如，如果我们确定威克森不能充分给澳得罗达传粉，就可以执行`delete(Eldorado,Wickson)`操作。对应Eldorado的链表表头在`Pollinizers[Eldorado]`中被找到，而且要从那里开始沿着链表向下行进，直到找到存放Wickson的单元并将其删除。

查找操作更是小菜一碟，只需要返回在合适的数组条目中找到的指针。例如，要对查询`lookup(Burbank,Pollinizers)`作出回应，只要返回链表`Pollinizers[Burbank]`就行了。

7.9.4　二元关系的散列表表示

我们可以使用只取决于有序对第一个组分的散列函数，将给定的二元关系R存储在散列表中。也就是说，有序对(a,b)会被放置在散列表元$h(a)$中，其中h是散列函数。请注意，这种安排与针对函数的安排是一模一样的，唯一的差异在于，对二元关系而言，一个散列表元中可能包含多个以给定的值a作为第一个组分的有序对，而对函数而言，它所含的这种有序对决不会超过一个。

要插入有序对(a,b)，就要计算$h(a)$，并对含有该成员的散列表元加以检查，以确保(a,b)尚未出现在其中。如果还没出现，就将(a,b)添加到该散列表元对应链表的末端。要删除(a,b)，就要先找到散列表元$h(a)$，然后查找该有序对，如果链表中存在该有序对，就将其删除。

要执行`lookup(a)`，就还是要先找到散列表元$h(a)$，然后沿着该散列表元对应的链表向下行进，收集所有在第一个组分为a的单元中出现的b。图7-24中为二元关系的链表表示编写的`lookup`函数也可以用于构成散列表表元的链表。

7.9.5　二元关系操作的运行时间

二元关系3种表示的性能与函数或词典上同样结构的性能差别不大。首先考虑链表表示。尽管还没有编写过用于插入和删除操作的函数，但我们应该能意识到这些函数会行遍整个链表，查找目标有序对，然后在找到它的地方停下。在长度为n的链表上，这样的查找平均会耗费$O(n)$

的时间，因为如果没找到这样的有序对，它肯定是扫描了整个链表，而如果找到了，它平均也要扫描链表的半数单元。

对查找操作来说，图7-24中的检测应该能说服我们，该函数所花的时间是 $O(1)$ 加上对链表尾部的递归调用耗费的时间。因此，如果链表长度为n，我们会执行n次调用，总共花费 $O(n)$ 的时间。

现在考虑一般化的特征向量。操作`lookup(a)`是最简单的。找到以a为下标的数组元素，可以在该元素处找到所需的答案——满足有序对 (a,b) 在该关系中的所有b组成的链表。我们甚至不必检验这些元素或复制它们。因此，在使用特征向量时，查找操作花的时间为 $O(1)$ 。

另一方面，插入和删除操作就没那么简单了。要插入 (a,b) ，可以相当容易地找到下标为a的数组元素，不过必须查找整个链表，以确保 (a,b) 尚未出现在其中。[①]这样做所需的时间与链表的平均长度成比例，也就是说，与关联某给定定义域值的值域值的平均数量成正比。我们将该参数称为m。另一种看待m的方式是，它是关系中有序对的总数量n除以不同定义域值的数量。如果假设任一链表与其他链表被查找的可能都是相同的，则执行插入或删除操作平均需要 $O(m)$ 的时间。

最后来考虑散列表。如果在关系中有n个有序对，并且散列表中有B的散列表元，就能预期平均每个散列表元中有n/B个有序对。不过，这里还是要引入参数m。如果存在n/m个不同的定义域值，那么至多有n/m个散列表元可以是非空的，因为对应有序对的散列表元只由定义域值决定。因此，不管B是多少，m是散列表元平均大小的下界。因为n/B也是下界，所以执行这3种操作其中之一所花的时间是 $O\big(\max(m,n/B)\big)$ 。

✦ 示例 7.31

假设有一个含1000个有序对的关系，这些有序对分布到100个定义域值中。那么每个定义域值会有10个值域值与之关联，也就是说$m=10$。如果使用1000个散列表元，也就是$B=1000$，那么m要大于n/B，也就是1，这样就可以预期我们实际可能查找的散列表元（因为表元编号为$h(a)$，其中a是关系中的某个定义域值）平均含有约10个有序对。事实上，每个散列表元中平均所含有序对数量要略多于这个值，因为不同的定义域值a_1和a_2在经过散列之后，得到的$h(a_1)$和$h(a_2)$可能恰巧是同一个散列表元。如果选择$B=100$，那么$m=n/B=10$，还是可以预期每个可能查找的散列表元含有约10个元素。正如刚刚提到的，实际数字可能要略大于10，因为可能出现两个或多个定义域值散列到同一散列表元的巧合。

7.9.6　习题

(1) 使用示例7.29中的数据类型编写函数，接受传粉者的值b以及由品种–传粉者有序对组成的链表作为参数，并返回由可以被b传粉的品种组成的链表。

(2) 使用示例7.29中的假设，编写用来处理品种-传粉者有序对的(a)插入；(b)删除程序。

(3) 为用示例7.30所述的向量数据结构表示的二元关系编写执行(a)插入；(b)删除；(c)查找操作的函数。在插入有序对时，不要忘了检查相同的有序对是否已经出现在该关系中。

(4) 设计散列表数据结构，用来表示构成本节中大量示例的传粉者关系。编写执行插入、删除和查找操作的函数。

---

① 也可以在不考虑该有序对是否已经出现的情况下直接插入该有序对，不过这样就会同时带来6.4节中讨论过的允许重复的链表表示所具有的优点和缺点。

(5) * 通过对链表L的长度进行归纳，证实`lookup`返回了满足有序对 (a,b) 在L中的所有元素b组成的链表，从而证明图7-24中的`lookup`函数可以正常工作。

(6) * 设计数据结构，使其执行插入、删除、查找和反向查找（`inverseLookup`）操作的平均时间可以达到 $O(1)$ 的水平。反向查找操作是接受值域元素，并找到与之关联的定义域元素。

(7) 在本节以及前面的几节中，我们定义了一些具有插入、删除和查找操作的新抽象数据类型。不过，这些操作与对词典的同名操作稍有差异。绘制表格，分别记下词典、函数（如7.8节所描述）和关系（如本节所描述）可能的抽象实现，以及支持这些抽象实现的数据结构。对每种实现，给出各操作的运行时间。

对函数和关系的“词典操作”

有序对的集合可以视为集合、函数或是关系。对每种情况来说，我们都已经定义了合适的插入、删除和查找操作。这些操作有着不同的形式。多数情况下，操作会同时取有序对的定义域元素和值域元素。不过，有时候只有定义域元素被用作参数。下表总结了这3种操作在使用中的差异。

	有序对集合	函　数	关　系
插入	定义域和值域	定义域和值域	定义域和值域
删除	定义域和值域	仅定义域	定义域和值域
查找	定义域和值域	仅定义域	仅定义域

7.10 二元关系的一些特殊属性

在本节中，我们将考虑某些实用的二元关系所具备的一些特殊属性。首先要定义一些基本属性：传递性、自反性、对称性与反对称性。这些结合起来就形成了几类常见的二元关系：偏序关系、全序关系和等价关系。

7.10.1 传递性

设R是定义域D上的二元关系。如果只要aRb和bRc为真，就有aRc也为真，就说关系R是传递的。图7-26展示了传递性这种属性。就像它在关系图中出现的那样，只要从a到b以及从b到c的虚线箭头出现在图中，那么从a到c的实线箭头也一定会出现在图中。谨记，传递性与本节中要定义的其他属性一样，都是关于整个集合的属性。只有3个特定的定义域元素满足该属性是不够的，声明的定义域D中所有的三元组a、b、c都必须满足。

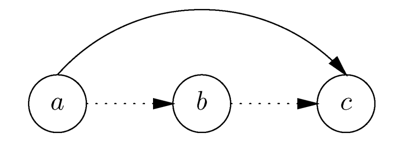


图7-26　传递性成立的条件要求如果aRb和bRc的弧在表示关系的图中出现，那么弧aRc也要出现

✦ 示例 7.32

考虑一下整数集**Z**上的 < 关系。也就是说，< 是满足a小于b的整数有序对 (a,b) 的集合。关

系 $<$ 是传递的，因为如果 $a<b$ 且 $b<c$，就可知 $a<c$。同样，整数上的关系 $\leqslant$、$>$ 和 $\geqslant$ 也都是传递的。这4种比较关系在实数集合上也同样具有传递性。

不过，考虑一下整数（或者是实数）上的 $\neq$ 关系。该关系就不具传递性。例如，设a和c都是3，并设b是5。这样 $a\neq b$ 与 $b\neq c$ 都为真。如果该关系是传递的，那么应该有 $a\neq c$。不过这就是说 $3\neq 3$，显然是错的。所以可以得出 $\neq$ 是不具传递性的。

再举个传递关系的例子，考虑一下 $\subseteq$，也就是子集关系。我们也许想将该关系视为所有满足S为T的子集的集合有序对(S,T)组成的集合，但想象一下，有这样的集合就会再次将我们引向罗素悖论。不过，假设有“全集”U，就可以设 $\subseteq_U$ 是集合有序对的结合

$$\{(S,T)\mid S\subseteq T且T\subseteq U\}$$

那么 $\subseteq_U$ 就是U的幂集 $\mathrm{P}(U)$ 上的关系，而我们可以将 $\subseteq_U$ 当作子集关系。

例如，设 $U=\{1,2\}$。那么 $\subseteq_{\{1,2\}}$ 就是由如图7-27所示的9个(S,T)有序对组成的。因此，$\subseteq_U$ 刚好含有满足第一个组分是第二个组分的子集（不一定是真子集），而且二者皆为$\{1,2\}$的子集的那些有序对。

不管全集U是什么，都很容易检验 $\subseteq_U$ 是传递的。如果 $A\subseteq B$ 而且 $B\subseteq C$，那么肯定有 $A\subseteq C$。原因在于，对A中的每个x，我们知道x也在B中，因为 $A\subseteq B$。因为x在B中，我们知道x也在C中，因为 $B\subseteq C$。因此A中的每个元素也都是C中的元素。所以 $A\subseteq C$。

S	T
$\varnothing$	$\varnothing$
$\varnothing$	{1}
$\varnothing$	{2}
$\varnothing$	{1，2}
{1}	{1}
{1}	{1，2}
{2}	{2}
{2}	{1，2}
{1，2}	{1，2}

图7-27　关系 $\subseteq_{\{1,2\}}$ 中的有序对

7.10.2　自反性

有些二元关系R还具有这样的属性，就是对声明的定义域中的每个元素a，R中都包含有序对 (a,b)，也就是都有aRa。如果这样的话，就说R是*自反的*。图7-28展示了某自反关系的图，其声明的定义域中每个元素上都有个循环。该图中除了这些循环外还可能有其他的箭头。不过，当前定义域中每个元素都有循环是不够的，必须是声明定义域中每个元素都有循环才行。

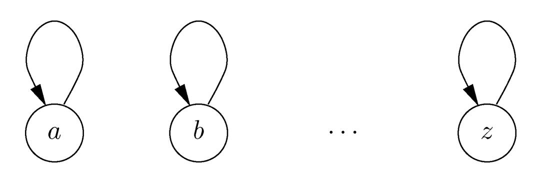


图7-28　自反关系R对其声明定义域中每个元素x来说都有xRx

✦ 示例 7.33

实数集合上的关系$\geqslant$就是自反的。对每个实数a而言，都有$a \geqslant a$。同样，$\leqslant$是自反的，而这两种关系在整数集合上也是自反的。不过，$<$和$>$就不是自反的，因为至少有一个a的值可以使$a>a$和$a<a$不成立，其实，对所有的a来说，$a>a$和$a<a$都是不成立的。

示例7.32中定义的子集关系$\subseteq_U$也是自反的，因为对任意集合A而言，都有$A \subseteq A$。不过，有着相似定义，包含满足$T \subseteq U$和$S \subset T$的有序对(S,T)的关系$\subseteq_U$——表示S是T的真子集的关系——就不是自反的。原因在于，$A \subset A$对某些A（事实上是对所有的A）来说不成立。

7.10.3 对称性与反对称性

设R是某二元关系。正如7.7节的习题(7)所定义的那样，R的*逆*是指将R中各有序对的组分调换位置后形成的新有序对组成的集合。也就是说，R的逆，记作R^{-1}，就是

$$\{(b,a) \mid (a,b) \in R\}$$

例如，$>$是$<$的逆，因为刚好当$b<a$时有$a>b$。同样，$\geqslant$是$\leqslant$的逆。

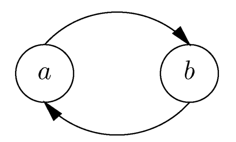


图7-29 对称性要求如果aRb，就也有bRa

如果R是它自己的逆，就说它是*对称的*。也就是说，如果只要aRb，就也有bRa，就说R是对称的。图7-29展示了在表示关系的图中对称性是什么样的。如果出现了向前的弧，就肯定还要有向后的弧。

如果只有$a=b$在时才有aRb和bRa都为真，我们就说R是反对称的。请注意，在反对称关系中，都不必有aRa对任意特定a来说为真。不过，反对称关系也可以是自反的。图7-30展示了在关系图中反对称的条件是怎样的。

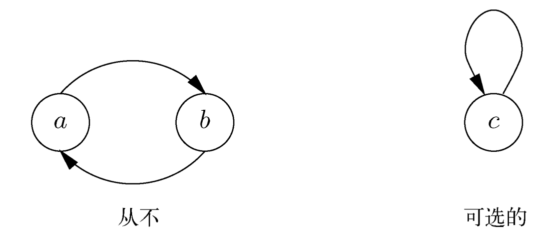


图7-30 反对称关系不能具有涉及两个元素的循环，不过单一元素上的循环是可以出现的

✦ 示例 7.34

整数集或实数集上的$\leqslant$关系就是反对称的，因为，如果$a \leqslant b$且$b \leqslant a$，就肯定有$a=b$。关系$<$也是反对称的，因为在任何条件下$a<b$和$b<a$都不可能同时成立。同样，$\geqslant$和$>$是反对称的，示例7.32中讨论的子集关系$\subseteq_U$也是。

不过，要注意到$\leqslant$不是对称的。例如，$3 \leqslant 5$，但$5 \leqslant 3$是不成立的。同样，上一段中提到的其他几种关系也都不是对称的。

整数上的$\neq$关系就是对称关系的一个例子。也就是说，如果$a \neq b$，就一定有$b \neq a$。

属性定义中的陷阱

正如前文已经指出的，属性的定义都是针对一般情况的，适用于定义域中的所有元素。例如，要让声明定义域D上的某关系R是自反的，就需要对每个$a \in D$都有aRa。aRa对某个a成立是不够的，而且说某个关系对某些元素自反而对另一些元素不自反也是说不通的。就算D中只有一个a让aRa不成立，也说明R不是自反的。因此，自反性可能取决于定义域，而且取决于关系R。

还有，像传递性——若aRb且bRc，则aRc——这样的条件具有“若A则B”的形式。请记住，要满足这样的命题，既可以让B为真，也可以令A为假。因此，对某个给定的三元组a、b和c，只要aRb为假，或bRc为假，或aRc为真，就满足传递性的条件。最极端的情况是，空关系是传递的、对称的而且反对称的，因为“若”的条件从不能满足。不过，空关系不是自反的，除非声明的定义域为$\varnothing$。

7.10.4　偏序和全序

偏序是传递且反对称的二元关系。如果除了传递性和反对称性之外，某关系能让每个定义域元素对都是可比的，就说该关系是全序关系。也就是说，如果R是全序的，而且a和b是其定义域中的任意两个元素，则要么aRb为真，要么bRa为真。请注意，每个全序关系都是自反的，因为可以设a和b是相同的元素，这样可比性的要求就告诉我们有aRa。

✦　示例 7.35

整数或实数上的算术比较≤和≥都是全序关系，因此也都是偏序关系。请注意，对任意的a和b来说，要么$a \leqslant b$，要么$b \leqslant a$，不过当$a = b$时刚好两者都成立。

算术比较 < 和 > 都是偏序关系而非全序关系。尽管它们是反对称的，不过不是自反的，也就是说$a<a$和$a>a$都不成立。

对应某个全集U的2^U上的子集关系$\subseteq_U$和$\subseteq_U$都是偏序关系。我们已经知道，它们是传递且反对称的。不过，只要U中至少有两个成员，这些关系就不是全序关系，因为这样一来就有不可比的元素了。例如，设$U = \{1,2\}$。那么$\{1\}$和$\{2\}$都是U的子集，但这两个集合之间谁也不是谁的子集。

大家可将全序关系R视作一个如图7-31所示的线性元素序列，其中只要对不同的元素a和b有aRb，a就出现在这条线上b的左侧。例如，如果R是整数上的≤关系，那么轴上的元素就是$\cdots, -2,-1,0,1,2,\cdots$。如果$R$是实数上的≤关系，那么这些点就对应实数轴上的点，就像这根轴是把无限长的尺子那样，如果实数x非负，那么x就是在0标记右侧x个单元处，而如果x为负，那么它就在0标记左侧$-x$个单元处。

如果R是偏序关系而非全序关系，还可以将定义域中的元素画成这样：如果aRb，那么a在b的左边。不过，因为可能存在不可比的元素，所以不一定能做到把所有元素画在一条轴上从而使关系R意味着“在左边”。

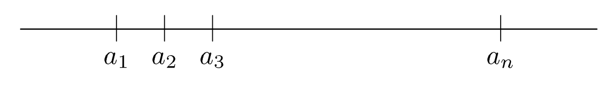


图7-31　表示$a_1, a_2, a_3, \cdots, a_n$上的全序关系的图

✦ 示例 7.36

图7-32展示了偏序关系$\subseteq_{\{1,2,3\}}$。我们已经将该关系绘成了简化图（reduced graph），在图中省略了可由传递性指出的弧。也就是说要有$S \subseteq_{\{1,2,3\}} T$，就要满足以下任一条件。

(1) $S = T$。

(2) 存在从S到T的弧。

(3) 从S到T之间有一条由两条或多条弧构成的路径。

例如，我们知道$\varnothing \subseteq_{\{1,2,3\}} \{1,3\}$，因为存在路径从$\varnothing$到$\{1\}$再到$\{1,3\}$。

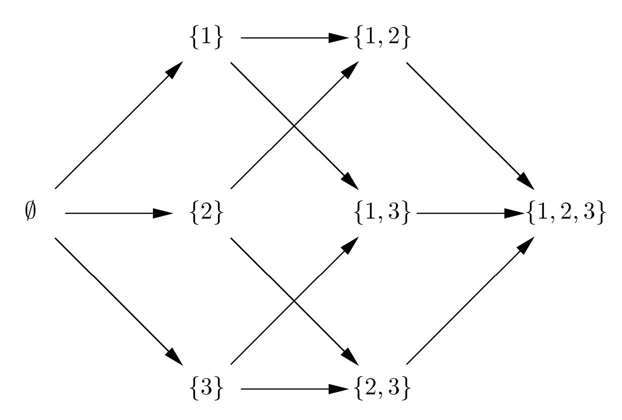


图7-32　表示偏序关系$\subseteq_{\{1,2,3\}}$的简化图

7.10.5　等价关系

等价关系是自反、对称且传递的二元关系。这种关系与之前的示例中看到的偏序关系和全序关系差别很大。事实上，偏序关系从不可能是等价关系，除非在声明的定义域为空，或者声明定义域中只有一个元素a而且该关系是$\{(a,a)\}$这样一些微不足道的情况下。

✦ 示例 7.37

像整数上的$\leqslant$这样的关系就不是等价关系。虽然它是传递且自反的，但它不是对称的。如果$a \leqslant b$，除非$a = b$，否则是不会有$b \leqslant a$的。

举个等价关系的例子，设R是由那些满足$a-b$是3的整数倍的整数有序对(a,b)组成的。比如，$3R9$，因为$3-9=-6=3\times(-2)$。还有$5R(-4)$，因为$5-(-4)=9=3\times3$。不过，(1,2)就不在R中，或者可以说“$1R2$不成立”，因为$1-2=-1$，它不是3的整数倍。可以按照如下方式展示R是等价关系。

(1) R是自反的，由于对任意整数a都有aRa，这是因为$a-a$为0，是3的整数倍。

(2) R是对称的。如果$a-b$是3的整数倍，比方说是$3c$，其中c为某整数，那么$b-a$就是$-3c$，因此也是3的整数倍。

(3) R是传递的。假设aRb而且bRc，也就是说，$a-b$是3的倍数，比方说是$3d$，而b-c也是3的倍数，比方说是$3e$。那么

$$a-c=(a-b)+(b-c)=3d+3e=3(d+e)$$

因此$a-c$也是3的倍数。由aRb和bRc得出aRc，这表示R是传递的。

再举个例子，设S是世界城市的集合，而T是由aTb定义的关系，其中a和b是由公路相连的，

也就是说，可以从a驾车到达b。因此，有序对（多伦多，纽约）是在T中，不过（檀香山，安克雷奇）就不在T中。可以说T是等价关系。

T是自反的，因为每个城市都是连接到它自己的。T也是对称的，因为如果a连接到b，那么b也连接到a。T还是传递的，因为如果a连接到b，且b连接到c，那么a是连接到c的，如果没有更短路径的话，可以通过b从a行驶到c。

7.10.6　等价类

另一种看待等价关系的方式是，它将子集的定义域分成了等价类。如果R是定义域D上的等价关系，那么可以将D分为等价类，使得下列命题成立。

(1) 每个定义域元素刚好在一个等价类中。

(2) 如果aRb，那么a和b在相同的等价类中。

(3) 如果aRb不成立，那么a和b在不同的等价类中。

✦　示例 7.38

考虑示例7.37中的关系R，其中当$a-b$是3的倍数时有aRb。一个等价类是刚好被3整除的整数的集合，也就是除以3余数为0的那些整数的集合。该类为$\{\cdots,-3,0,3,6,\cdots\}$。第二个是除以3时余数为1的整数的集合，也就是$\{\cdots,-2,1,4,7,\cdots\}$。最后一个类是除以3时余数为2的整数的集合，该类为$\{\cdots,-1,2,5,8,\cdots\}$。这些类将整数集划分成3个不相交的集合，如图7-33所示。

请注意，当两个整数除以3的余数相同时，它们的差就能被3整除。例如，$14=3\times4+2$而$5=3\times1+2$，因此$14-5=3\times4-3\times1+2-2=3\times3$，于是可知$14R5$。另一方面，如果两个整数除以3的余数不同，它们的差就肯定不能被3整除。因此，来自不同等价类的整数（比如5和7）之间，就不具备R关系。

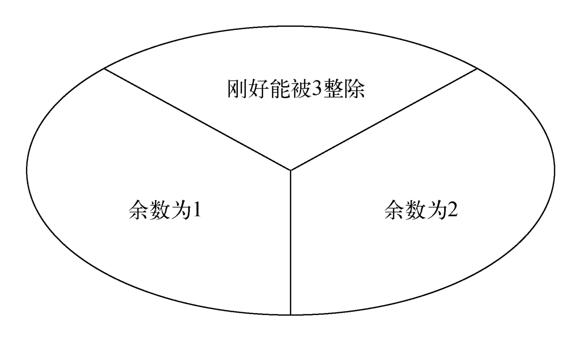


图7-33　整数上的关系“差能被3整除”相应的等价类

要为等价关系R构建等价类，设$class(a)$是满足aRb的元素b的集合。例如，如果等价关系是示例7.37中我们称为R的那个，那么$class(4)$就是除以3时余数为1的整数的集合，也就是说$class(4)=\{\cdots,-2,1,4,7,\cdots\}$。

请注意，如果让a对定义域的各元素而言是不同的，通常会多次得到同样的类。其实，当有aRb时，就有$class(a)=class(b)$。要知道为什么，可以假设c在$class(a)$中。则根据类的定义有aRc。因为给定了aRb，根据对称性有bRa。而根据传递性，由bRa和aRc可以得出bRc。而bRc就说明c在$class(b)$中。因此，$class(a)$中的每个元素都在$class(b)$中。因为同样的推理告诉我们，只要aRb，那么$class(b)$中的每个元素也都在$class(a)$中，所以我们可以得出结论：$class(a)$和$class(b)$是相同的。

不过，如果$class(a)$和$class(b)$不同，则这些类不可能有相同的元素。作相反的假设，那么就

肯定有某个c同时在$class(a)$和$class(b)$中。而根据之前的假设，知道有aRc和bRc。根据对称性，有cRb。根据传递性，可由aRc和cRb得到aRb。不过我们刚证明了，只要aRb成立，则$class(a)$和$class(b)$是相同的。而这里假设这些类是不同的，因此就得出了矛盾。所以，假设的出现在$class(a)$和$class(b)$的交集中的元素c不可能存在。

还要看到：每个定义域元素都在某个等价类中。特别要说的是，a总是在$class(a)$中，因为自反性告诉我们有aRa。

我们现在就可以得出结论，等价关系将其定义域划分为不相交的等价类，而且将每个元素刚好放在一个类中。示例7.38就展示了这一现象。

7.10.7　关系的闭包

对二元关系的常见运算还有一种，就是取某个不具有自反性（或对称性、传递性）的集合，在为其添加尽可能少的有序对后使得新形成的关系具有自反性（或对称性、传递性）。得到的关系就称为原关系的自反（或对称、传递）闭包。

✦　示例 7.39

我们在图7-32中讨论过简化图。虽然表示的是传递关系$\subseteq_{\{1,2,3\}}$，但是只画出了与该关系中有序对的某个子集对应的弧。不过通过应用传递法则推断出新的有序对，直到不能推断出新的有序对，就可以重建完整的关系。例如，我们看到存在有序对$(\{1\},\{1,3\})$和$(\{1,3\},\{1,2,3\})$相对应的弧，因此传递法则就告诉我们有序对$(\{1\},\{1,2,3\})$也肯定在该关系中。而该有序对与有序对$(\varnothing,\{1\})$一起，又说明$(\varnothing,\{1,2,3\})$也在该关系中。除此之外，还必须加上“自反的”有序对(S,S)，其中S是$\{1,2,3\}$的各个子集。这样一来，就重建了关系$\subseteq_{\{1,2,3\}}$中的所有有序对。

另一种实用的闭包运算是*拓扑排序*，我们接受某个偏序，并向其添加元组，直到它成为全序。尽管二元关系的传递闭包是唯一的，但常常有多个全序包含某一给定的偏序。我们将在第9章中了解到一种特别高效的拓扑排序算法。现在，先考虑一个展示拓扑排序实用性的例子。

✦　示例 7.40

人们常将生产过程中必须执行的一系列任务表示为一套必须服从的“优先级”。举个简单的例子，在给左脚穿鞋之前必须先给左脚穿上袜子，而在穿上右脚的鞋之前要先穿上右脚的袜子。不过，这其中没有其他必须遵守的优先级了。我们可以用由两个有序对(左袜，左鞋)和(右袜，右鞋)组成的集合来表示这些优先级。该集合是个偏序。

可以将该集合扩展为6个不同的全序。其中一个全序是先穿好左脚的鞋袜，该关系是含以下10个有序对的集合。

(左袜，左袜)　(左袜，左鞋)　(左袜，右袜)　(左袜，右鞋)
(左鞋，左鞋)　(左鞋，右袜)　(左鞋，右鞋)
(右袜，右袜)　(右袜，右鞋)
(右鞋，右鞋)

可将该全序视作如下线性排列

$$左袜 \to 左鞋 \to 右袜 \to 右鞋$$

先穿好右脚的鞋袜有着与之相似的过程。

由原始的偏序还可以得到其他4种全序，其中我们要先穿袜子再穿鞋，它们可由以下线性排列表示：

左袜→右袜→左鞋→右鞋
左袜→右袜→右鞋→左鞋
右袜→左袜→左鞋→右鞋
右袜→左袜→右鞋→左鞋

闭包的第三种形式是找到含有某给定关系的最小等价关系。例如，公路图表示的关系是由公路路段直接连接而不含中间城市的城市对组成的。要确定由公路连接的城市，可以利用自反性、传递性和对称性推断出由某些基础道路序列连接的城市对。闭包的这种形式称为找出图中的“连通分支”（connected component），我们将在第9章讨论一种解决该问题的高效算法。

7.10.8　习题

(1) 给出对某一声明定义域自反，但对另一声明定义域不自反的关系。请记住，对作为某关系R可能的定义域的D而言，D必须包含出现在R的有序对中的每个元素，但它还可以包含更多的元素。

(2) **关系$\subseteq_{\{1,2,3\}}$中有多少个有序对？考虑一般的情况，如果U有n个元素，那么$\subseteq_U$中有多少个有序对？提示：试着从元素较少的情况猜测该函数，比如含两个元素的情况下有9个有序对，如图7-27所示。然后通过归纳证明自己的猜测是正确的。

(3) 考虑定义域在4字母字符串上的二元关系R，它是由sRt定义的，其中t是由字符串s的字母向左循环移动一位形成的。也就是说，$abcdRbcda$，其中a、b、c、d都是单独的字母。确定R是否为(a)自反的；(b)对称的；(c)传递的；(d)偏序，和（或）(e)等价关系。为每种情况给出简要论证或是反例。

(4) 考虑习题(3)中的4字母字符串定义域。设S是应用0次或多次R组成的二元关系。因此，$abcdSabcd$，$abcdSbcda$，$abcdScdab$，且$abcdSdabc$。换句话说，字符串与它经过任意循环位移后形成的字符串具有S关系。对关系S回答习题(3)中提出的5个问题，并且每种情况都要给出论证。

(5) * 以下“证明”有何错误？

（非）定理：如果二元关系R是对称且传递的，那么R是自反的。

（非）证明：设x是R定义域中的某个成员，取某个满足xRy的y。根据对称性，有yRx。而根据传递性，xRy和yRx可以得出xRx。因为x是R定义域的任一成员，所以证明了xRx对R定义域中的每个元素都成立，也就“证明”了R是自反的。

(6) 给出声明定义域为{1,2,3}，具有如下属性的二元关系的例子。

(a) 自反且传递，但不对称。

(b) 自反且对称，但不传递。

(c) 对称且传递，但不自反。

(d) 对称且反对称。

(e) 自反，传递，而且是全函数。

(f) 反对称，而且是一一对应。

(7) * 如果为关系$\subseteq_U$使用简化图，其中集合U有n个元素，那么与使用完全图相比要节省多少条弧？

(8) 当U只有一个元素时，(a) $\subseteq_U$和(b) $\subset_U$是否为偏序或全序？当U中没有元素时呢？

(9) * 从$n=1$开始，通过对n的归纳证明，如果有n个有序对a_0Ra_1、a_1Ra_2、⋯、$a_{n-1}Ra_n$，而且如果R是传递的关系，那么有a_0Ra_n。也就是要证明，如果表示传递关系的图中存在任一路径，就存在一条从该路径开头到该路径结尾的弧。

(10) 找出包含有序对(a,b)、(a,c)、(d,e)和(b,f)的最小等价关系。

(11) 设R是整数集上满足如下条件的关系，若a和b是互不相同的而且有除了1之外的公约数，则aRb。

确定R是否为(a)自反的；(b)对称的；(c)传递的；(d)偏序和（或）(e)等价关系。

(12) 存在某树T所有节点上的关系 R_T，其中当且仅当在树T中a是b的祖先时有 aR_Tb，针对该关系重复习题(11)中的练习。

(13) 存在某树T所有节点上的关系 S_T，其中当且仅当在树T中a在b的左侧时有 aS_Tb，针对该关系重复习题(12)中的练习。

7.11　无限集

人们在计算机程序中要实现的所有集合都是有限的，如果这些集合不是有限的，就没法将它们存储在计算机的内存中。而在数学中，很多集合（比如整数集或实数集）都是无限的。这些观点似乎直观清晰，不过有限集和无限集到底有何区别呢？

有限集和无限集之间的区别是相当令人惊讶的。有限集的元素数量与它任一真子集的元素数量都不同。回想一下，在7.7节中，我们说过可以利用两个集合间一一对应的存在得出它们是等势的（equipotent），也就是说，它们有着相同数量的成员。

如果取一个如$S = \{1,2,3,4\}$这样的有限集及其任意真子集，如$T = \{1,2,3\}$，那么在这两个集合间没办法找到一一对应。例如，可以把S中的4映射到T中的3，把S中的3映射到T中的2，把S中的2映射到T中的1，但接着就找不出T中的成员来和S中的1相关联。其他建立从S到T的一一对应的尝试也一定同样失败。

大家直观上可能会认为这一点对任意集合来说都应该成立，一个集合在丢掉其中一个或多个元素后怎么可能还具有相同的元素数呢？考虑一下自然数（非负整数）集**N**和**N**去掉0后得到的真子集，称该集合为$\mathbf{N}-\{0\}$，$\{1,2,3,\cdots\}$。那么考虑一下从**N**到$N-\{0\}$的一一对应F，其中$F(0)=1$，$F(1)=2$，一般来讲，$F(i)=i+1$。

惊人的是，F是从**N**到$\mathbf{N}-\{0\}$的一一对应。对**N**中的每个i，至多有一个j满足$F(i)=j$，所以F是个函数。其实，刚好就有一个这样的j，即$i+1$，使得一一对应的定义中的条件(1)（见7.7节）得到满足。对$\mathbf{N}-\{0\}$中的每个j，存在某个i满足$F(i)=j$，也就是，$i=j-1$。因此一一对应的定义中的条件(2)也得到满足。最后，在**N**中不存在两个不同的数字i_1和i_2使得$F(i_1)$和$F(i_2)$都为j，因为那样的话i_1+1和i_2+1都为j，这样一来就得出$i_1=i_2$，进而就得出F是**N**与其真子集$\mathbf{N}-\{0\}$之间一一对应的结论。

无限酒店

为了帮助大家理解从0开始和从1开始有着同样多的数字，可以想象一家酒店，它有着无限个房间，分别编号为0、1、2，等等。对任意整数而言，都存在一个以该整数作为房号的房间。在某一特定时间，每个房间里都会有一名顾客。一只袋鼠来到前台开房间。前台接待告诉它：“我们这里不接待袋鼠。”等一下，这跑题了。事实上，前台接待按照如下方式给袋鼠腾出了房间。他让0号房间的客人住进1号房，让1号房的客人住进2号房，等等。所有的旧客都还是有一间房可住，而现在0号房是空房了，而这只袋鼠就住进了0号房。这种“戏法”之所以能奏效，是因为从1开始编号的房间与从0开始编号的房间其实是同样多的房间。

7.11.1　无限集的正式定义

数学家们认可的定义是，无限集是指自身与其至少一个真子集之间存在一一对应的集合。

在一些极端例子下，无限集和其某个真子集之间可以存在一一对应关系。

✦ 示例 7.41

自然数集合与偶自然数集合是等势的。设$F(i)=2i$。那么F就是一一对应，它将0映射到0，1映射到2，2映射到4，3映射到6，而一般来讲，就是将每个自然数映射到一个唯一的自然数，它的两倍。

同样，**Z**和**N**是同样大小的集合，也就是说，非负整数和负整数一起，与非负整数是一样多的。设对所有的$i \geqslant 0$，有$F(i)=2i$，并设对所有的$i<0$，有$F(i)=-2i-1$。那么0映射到0，1映射到2，−1映射到1，2映射到4，−2映射到3，等等。每个整数都被映射到一个唯一的非负整数，其中负整数映射为奇数，而非负整数则映射为偶数。

更让人咋舌的是，自然数对组成的集合与N本身也是等势的。要知道这样的一一对应是如何构建起来的，可以考虑一下图7-34，其中展示了$\mathbf{N}\times\mathbf{N}$中的有序对分布在一个无限的方阵中。我们根据有序对中组分的和来确定它们的次序，而对那些组分的和相等的有序对，则根据其第一个组分的大小确定次序。这一次序始于(0,0)、(0,1)、(1,0)、(0,2)、(1,1)、(2,0)、(0,3)、(1,2)，等等，如图7-34所示。

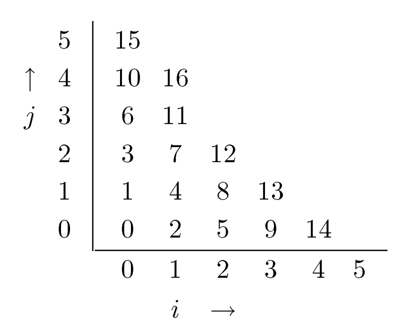


图7-34　为自然数对排序

现在，这些自然数对有了先后次序。原因在于，对任意自然数对(i,j)，和比其小的自然数对的数量是有限的，而和相同情况下值i更小的自然数对的数量也是有限的。其实，我们可以计算自然数对(i,j)在这一次序中的位置，就是$(i+j)\ (i+j+1)/2+i$。也就是说，我们的一一对应是将自然数对(i,j)与唯一的自然数$(i+j)\ (i+j+1)/2+i$关联起来。

请注意，一定要谨慎选择为有序对排序的方式。假设在图7-34中按行排序，那么永远都没法到达第二行或更高行的自然数对，因为每一行中都有无数个自然数对。同样，按列排序也是行不通的。

集合不是有限的，就是无限的

乍一看，可能会出现不那么有限和不那么无限的事物。例如，当谈论链表时，对链表的长度未作限制。而只要在程序的执行中创建了链表，它就具有了有限的长度。因此，可以作出如下区分。

(1) 每个链表的长度都是有限的，也就是说，它的单元数是有限的。

(2) 链表的长度可能是任何非负整数，而链表可能长度的集合是无限的。

无限集的正式定义是很有意思的，不过这一定义可能不符合我们对无限集的直觉认识。例如，我们可能觉得无限集是对每个整数n而言，包含至少n个元素的集合。好在可以证明这一属性是每个由正式定义可知无限的集合都具备的，这一证明过程又要用到归纳法。

命题 $S(n)$。如果I是有限集，那么I具有一个含n个元素的子集。

依据。设$n=0$，显然有$\varnothing\subseteq 1$。

归纳。假设对某个$n\geqslant 0$有$S(n)$。要证明I有一个含$n+1$个元素的子集。根据归纳假设，I有一个含n个元素的子集T。根据无限集的正式定义，存在某个真子集$J\subset I$，以及从I到J的一一对应f。设a是$I-J$中的元素，因为J是个真子集，所以a肯定是存在的。

考虑R，T在f下的镜像，也就是说，若$T=\{b_1,\cdots,b_n\}$，则$R=\{f(b_1),\cdots,f(b_n)\}$。因为f是一一对应，则$f(b_1),\cdots,f(b_n)$各不相同，所以$R$的大小也为$n$。因为$f$是从$I$到$J$的，所以每个$f(b_k)$都在$J$中，也就是说$R\subseteq J$。因此，$a$不可能在$R$中。这样一来，$R\cup\{a\}$就是$I$含$n+1$个元素的子集，这证明了$S(n+1)$。

集合的基数

如果存在从S到T的一一对应，就定义两个集合S和T是等势的（大小相等）。等势是在任意由集合组成的集合上的等价关系，我们将这一点留作本节习题。集合S所属的等价类就称作S的基数。例如，空集属于它自身的等价类，可以用基数0来标识该类。含有集合$\{a\}$（其中a为任意元素）的类是基数1，而含集合$\{a,b\}$的类是基数2，等等。

含**N**的类是“整数的基数”，通常称为阿列夫零（aleph-0），而该类中的集合都是可数集。实数的集合属于另一个通常被称为连续统的等价类。其实，不同的无限基数有无数个。

7.11.2 可数集与不可数集

由示例7.41，我们可能会认为所有无限集都是等势的。我们已经看到整数的集合**Z**以及非负整数的集合**N**是同样大小的，还有一些直觉上讲“似乎”比**N**小的集合也与它大小相同。因为我们在示例7.41中看到，自然数对是与**N**等势的，而非负有理数也是与自然数等势的，因为有理数是由其分子和分母组成的自然数对。同样，可以证明（非负和负）有理数与整数是等势的，因此也就与自然数是等势的。

对任意集合S而言，如果存在从S到**N**的一一对应，就说该集合是可数的。这里用到术语“可数的”是说得通的，因为S肯定有一个与0对应的元素，一个与1对应的元素，等等，所以可以“数”S的成员。我们之前说过的，整数、有理数、偶数，以及自然数对的集合，都是可数集。还有很多其他的可数集，我们在这里把对合适的一一对应的探索留作练习。

不过，也存在不可数的无限集。特别要指出的是，实数就是不可数的。其实，可以证明从0到1之间的实数要比自然数多。论证的关键在于，0到1之间的实数都可以表示为无限长度的小数。我们为小数点右侧的位标记上0、1等编号，如果从0到1之间的实数是可数的，那么可以将它们标记为r_0、r_1，等等，然后就可以将这些实数排列在一个无限的方阵表格中，如图7-35所示。在假设的从0到1的所有实数的排列中，$\pi/10$被分配到第0行，5/9被分配到第1行，5/8被分配到第2行，4/33被分配到第3行，等等。

不过，可以证明图7-35并不能真正表示0到1这个范围内所有实数的列表。我们的证明是被

称为对角化的一类过程，要使用表的对角线创造出一个不可能在该实数列表中的值。假设创造一个新实数r，其十进制表示为$0.a_0a_1a_2$。第i位的值a_i，取决于对角线上的第i个数字，也就是在第i个实数的第i位找到的值。如果该值是0到4，就设$a_i = 8$。如果对角线上第i个位置是5到9，那么$a_i = 1$。

位置

实数↓	0	1	2	3	4	5	6	…
0	3	1	4	1	5	9	2	…
1	5	5	5	5	5	5	5	…
2	6	2	5	0	0	0	0	…
3	1	2	1	2	1	2	1	…
4								

图7-35　假设实数是可数的，表示实数的假想表格

✦ 示例 7.42

给定如图7-35所示的部分表格，我们的实数r是从0.8118…开始的。要知道原因，请注意，0号实数0号位置的值是3，所以$a_0 = 8$。1号实数1号位置的值是5，所以$a_1 = 1$。接下来，2号实数2号位置的值是5而3号实数3号位置的值是2，所以接下来的两位数字是18。

我们的主张是，即便假设所有从0到1的实数都在该表中，r也不会出现在这一假想的实数列表中。假设r是r_j，与第j行关联的实数。考虑r与r_j的差d。r的十进制展开第j位数字为a_j，我们知道该值是具体选择的，从而与r_j第j个位置的数字存在至少为4至多为8的差。因此，第j个位置对d的贡献在$4/10^{j+1}$到$9/10^{j+1}$之间。

第j位之后的所有位置对d的贡献加起来不会超过$1/10^{j+1}$，因为这就是r和r_j那些位置上一个全为0而另一个全为9时的差。因此，j及j之后的各个位置对d的贡献在$3/10^{j+1}$到$9/10^{j+1}$之间。

最后，在第j位之前的位置中，r和r_j要么是相同的，在这种情况下，前$j-1$位对d的贡献为0；要么就是r和r_j之间至少存在$1/10^j$的区别。不管哪种情况，我们都可以看到d不会为0。因此，r和r_j不可能是同一个实数。

这样就可以得出r不在该实数列表中的结论。因此，我们假设的这种从非负实数到从0到1之间实数的一一对应其实不是一对一的。这样就证明了，在0到1的范围内至少存在一个实数r不与任何整数相关联。

7.11.3　习题

(1) 证明等势是一种等价关系。提示：难点在于传递性，要证明如果存在从S到T的一一对应f，而且存在从T到R的一一对应g，就存在从S到R的一一对应。该函数是f和g的复合函数，也就是将S中的x变为R中的$g\big(f(x)\big)$的函数。

(2) 在图7-34所示的有序对次序中，编号为100的有序对是哪个？

(3) * 证明以下集合是可数的（在它们和自然数之间存在一一对应）。

(a) 完全平方数的集合。

(b) 自然数三元组(i,j,k)的集合。

(c) 2的乘方的集合。

(d) 自然数有限集组成的集合。

(4) ** 证明自然数的幂集**P**(**N**)与实数有着相同的基数，也就是说，存在从**P**(**N**)到0至1这一范围的实数的一一对应。请注意，这一结论与习题(3)的(d)小题并不矛盾，因为现在讨论的是整数的有限集和无限集，而我们只能为有限集计数。提示：以下构造几乎能行得通了，不过还需要进行修正。考虑一下任意自然数集合的特征向量。该向量是有限的0和1组成的序列。例如，{0，1}的特征向量是1100…，而含奇数个自然数的集合的特征向量则是010101…。如果在特征向量前加上小数点，就得到了0到1之间的二进制小数，它是表示实数的。因此，每个集合都可以转换为0到1范围内的实数，而且通过将二进制表示转换成特征向量，该范围内的每个实数都可以与一个集合相关联。这种关联不是一一对应的原因在于，某些实数可能会有两种二进制表示。例如，0.11000…和0.10111…都表示实数3/4。不过，这两个二进制小数对应的特征向量表示的是不同的集合，前者表示{0，1}，而后者则表示除了1之外的所有整数组成的集合。大家可以修改这种构造以定义一一对应。

(5) ** 证明：从0到1范围内的实数组成的有序对到该范围的实数间存在一一对应。提示：要模仿图7-34中的表格是不可能的。不过，我们可以取某个实数对，比方说是(r，s)，然后将表示r和s的无限小数集合起来，形成唯一的新实数t。t与r和s之间不是以简单的算术表达式相关联的，不过从t，可以唯一地恢复恢复r和s。大家必须找出一种方式，从r和s的十进制展开构建t的十进制展开。

(6) ** 证明：只要集合S包含所有整数大小0，1，…的子集，该集合就是符合“无限集”正式定义的无限集，也就是说，S与它的一个真子集间存在一一对应。

7.12 小结

大家应该从本章中了解到了以下要点。

- 集合的概念对数学与计算机科学来说都是基础。
- 集合的常见运算包括可以用文氏图直观呈现的并集、交集和差集运算。
- 代数法则可用于处理和简化涉及集合与集合运算的表达式。
- 链表、特征向量和散列表提供了3种表示集合的基本方式。链表提供了适用于最多集合运算的最佳灵活性，但并非总是最高效的。特征向量对某些集合运算而言有着最快的速度，但只能用于全集规模较小的情况。散列表是通常被选用的方式，兼具表示的经济性与访问的迅速性。
- （二元）关系是有序对的集合。函数是对某给定的第一个组分而言至多有一个元组的关系。
- 两个集合间的一一对应关系是个函数，它会给第一个集合中的各个元素关联上第二个集合中的唯一元素，反之亦然。
- 二元关系具有一些重要属性，其中自反性、传递性、对称性和反对称性属于最重要的。
- 偏序、全序和等价关系是二元关系的重要特例。
- 无限集是指那些与其某一真子集间存在一一对应关系的集合。
- 一些无限集是“可数的”，也就是说，它们与整数间存在一一对应的关系。另外一些无限集，比如实数，是不可数的。
- 在本章中定义的集合和关系上的数据结构与运算还会在本书剩下的部分以多种不同的方式使用。

7.13　参考文献

Halmos [1974]很好地介绍了集合论。散列技术最早是在20世纪50年代诞生的，而Peterson [1957]涵盖了早期的散列技术。Knuth [1973]和Morris [1968]包含了更多有关散列技术的材料。Reingold [1972]讨论了基本集合运算的计算复杂度。无限集的理论是由Cantor [1915]提出的。

Cantor, G. [1915]. “Contributions to the founding of the theory of transfinite numbers,” reprinted by Dover Press, New York.

Halmos, P. R. [1974]. *Naive Set Theory*, Springer-Verlag, New York.

Knuth, D. E. [1973]. *The Art of Computer Programming*, Vol. III, *Sorting and Searching*, Addison-Wesley, Reading, Mass.

Morris, R. [1968]. “Scatter storage techniques,” *Comm. ACM* **11**:1, pp. 35–44.

Peterson, W. W. [1957]. “Addressing for random access storage,” *IBM J. Research and Development* **1**:7, pp. 130–146.

Reingold, E. M. [1972]. “On the optimality of some set algorithms,” *J. ACM* **19**:4,pp. 649–659.

第 8 章

关系数据模型

计算机最为重要的一项应用就是存储和管理信息。信息的组织方式对访问和管理信息的容易程度有着深刻的影响。也许最简单而最万能的信息组织方式就是将其存储在表中。

关系模型就是这一概念的核心：数据被组织成称为“关系”的二维表集合。我们还可以将关系模型视作第7章讨论的集合数据模型的一般化，是将二元关系扩展到任意元的关系。

之所以最初要研发关系数据模型，是因为要用于*数据库*，即长时间存储在计算机系统中的信息，并用于*数据库管理系统*，即让人们可以存储、访问和修改这些信息的软件。数据库仍然是我们理解关系数据模型的重要动机。现在它们不仅存在于最初的那些大规模应用中，比如航空订票系统或银行系统，而且可以应用于桌面计算机，处理一些个人活动，诸如维持支出记录、作业成绩，此外还有其他很多用途。

除了数据库系统，其他类型的软件也可以很好地利用信息表，而关系数据模型有助于我们设计这些表，并研究出高效访问这些信息表所需的数据结构。例如，这样的表可被编译器用来存储与程序中变量有关的信息，记录它们的数据类型以及定义它们的函数。

8.1　本章主要内容

本章有3个相互交织的主题，首先要向大家介绍的是使用关系模型的信息结构的设计，我们会看到如下内容。

- 信息表，称作“关系”，是强大而灵活的信息表示方式（8.2节）。
- 设计过程中的重要环节是在不引入“冗余”，即某一事实重复若干次的情况下，选择可一起存储在表中的“属性”或所描述对象的属性（8.2节）。
- 表中的列是用属性命名的。表（或关系）的“键”是其值能唯一确定表中整行值的属性构成的集合。知道表的键有助于设计表示表的数据结构（8.3节）。
- 索引是有助于我们迅速检索或更改表中信息的数据结构。如果想高效地操作表，明智地选择索引是至关重要的（8.4节、8.5节和8.6节）。

第二个主题是数据结构加速数据访问的方式，在这部分内容中我们会了解到如下内容。

- 诸如散列表这样的主索引结构将表中各行安排在计算机的内存中，合适的结构可以提高诸多操作的效率（8.4节）。
- 辅助索引提供了额外的结构，而且有助于高效执行其他操作（8.5节和8.6节）。

第三个主题是一种非常高级的表示“查询”的方式，其中查询是指与表集合中的信息有关的问题，这部分有以下要点。

- 关系代数是一种强大的表示法，可以在不给出运算执行细节的情况下表示查询（8.7节）。
- 关系代数的运算符可以用本章讨论的数据结构来实现（8.8节）。
- 为了迅速得出用关系代数表示的查询的解答，通常有必要对它们加以“优化”，也就是说，使用代数法则将一个表达式转换成有着更快求值策略的等价表达式。我们将在8.9节中了解一些这样的技巧。

8.2　关系

7.7节介绍的“关系”的概念是元组的集合。关系中的每个元组都是一列组分，而每个关系都具有固定的元数，它表示每个元组中所含组分的数量。尽管我们主要研究了二元关系，也就是元数为2的关系，但也说过其他元数的关系不仅存在，而且相当实用。

关系模型中用到的“关系”的概念与关系的集合论定义是紧密相关的，但在某些细节上存在差异。在关系模型中，信息被存储在如图8-1所示的表中。图中的表所表示的数据可能存储在教务老师的计算机中，是与课程、选择这些课程的学生以及他们所取得的成绩有关的信息。

表中的列都被给定了名称，这些名称就叫做属性（attribute）。在图8-1中，属性分别有课程（Course）、学号（StudentID）和成绩（Grade）。

课　程	学　号	成　绩
CS101	12345	A
CS101	67890	B
EE200	12345	C
EE200	22222	B+
CS101	33333	A−
PH100	67890	C+

图8-1　信息表

作为集合的关系与作为表的关系

在关系模型中，正如我们在7.7节中对集合论关系的讨论那样，关系也是元组的集合。因此，表中各行排列的次序是不重要的，可以随意重新排列表中各行而不会改变表的值，就像重新排列集合中元素的次序而不改变集合的值那样。

表的每一行中各组分的次序则是关键的，因为不同的列有着不同的名称，而且每个组分所表示的项，必须具有该列标题所指示的类型。不过，在关系模型中，可以将一整列连同标题的名称一起改变次序，这样就能保持该关系不发生变化。数据库关系在这方面与集合论关系不同，不过我们很少会重新排列表中的列，因此可以保留同样的术语。为了避免疑问，本章中的术语“关系”总是具有数据库的含义。

表中各行被称为元组，且表示基本事实。第一行，(CS101，12345，A)，表示学号12345的学生在课程CS101中得了A。

表包含两个方面。

(1) 列名的集合；

(2) 包含信息的行。

术语“关系”指的是后者，也就是行的集合。每一行表示关系的一个元组，而且这些行在表中出现的次序是无关紧要的。相同表中不存在各列的值全部相同的两行。

第(1)项，列名（属性）的集合被称为关系的*模式*（scheme）。属性在模式中出现的次序无关紧要，不过为了正确地写出元组，需要知道属性与表中的列之间的对应关系。我们通常会使用模式作为关系的名称。因此，图8-1中的表通常称为“课程–学号–成绩”关系。此外，还可以用首字母缩写*CSG*来为该关系命名。

8.2.1　关系的表示

就像集合那样，用数据结构表示关系的方式也多种多样。表的各行就应该是结构体，其中各个字段与各列名相对应。例如，图8-1所示关系中的元组可以表示为如下类型的结构体。

```
struct CSG {
    char Course[5];
    int StudentId;
    char Grade[2];
};
```

表本身可以从多种方式中任选其一来表示，比如

(1) 该类型结构体组成的数组。

(2) 该类型结构体组成的链表，其中还要有链接链表各单元的`next`字段。

此外，也可以将一个或多个属性视为该关系的“定义域”，而将其余属性视作“值域”。例如，图8-1中的关系可以被看作从定义域“课程”到由“学号–成绩”有序对组成的值域的关系。然后可以按照7.9节中讨论过的二元关系的模式，将该关系存储在散列表中。也就是说，我们会散列“课程”的值，而存储在散列表元中的元素是“课程–学号–成绩”三元组。我们将从8.4中起更为详细地讲解表示关系的数据结构的这一问题。

8.2.2　数据库

关系的集合称为*数据库*。在为某些应用设计数据库时，首先要做的就是决定待存储的数据该如何安排在表中。与所有设计问题一样，数据库的设计也是个业务需求和判断的问题。在接下来的示例中，我们将扩展这里涉及课程的教务数据库应用，而且要揭示优秀数据库设计的一些原则。

数据库最强大的那些操作涉及将若干关系用来表示协调的数据类型。通过建立恰当的数据结构，可以高效地从一个关系跳转到另一个关系，从而从数据库中获取一些无法从单个关系发现的信息。与关系间“导航”相关的数据结构和算法将在8.6节和8.8节中加以介绍。

数据库中各关系的模式组成的集合就是数据库的*模式*。要注意数据库模式（它告诉我们与数据库中信息组织方式有关的信息）与各关系中元组的集合（数据库中存储的实际信息）之间的区别。

✦　示例 8.1

我们为图8-1中具有模式{课程，学号，成绩}的关系补充4个其他的关系，它们的模式和直观意义如下。

(1) {学号，姓名，地址，电话}。学生的学号出现在元组的第一个组分，而姓名、地址和电话号码分别出现在第二、第三和第四个组分中。

(2) {课程，前提}。该元组第二个组分表示的课程，是选修第一个组分所表示课程的前提。

(3) {课程，日子，时刻}。第一个组分表示的课程，是在由第二个组分指定的日子，第三个组分给出的时刻上课。

(4) {课程，教室}。第一个组分表示的课程是在第二个组分表示的教室上课。

这4个模式，加上之前提到的{课程，学号，成绩}模式，就构成了用于本章示例的数据库模式。我们还需要一个表示数据库可能的“当前值”的示例。图8-1给出了“课程–学号–成绩”关系的一个例子，而对应其他4个模式的示例关系如图8-2所示。请记住，这些关系远比我们在现实中遇到的关系简短，在这里只是需要提供一些对应这些模式的样本元组而已。

学　号	姓　名	地　址	电　话
12345	C.Brown	12 Apple St.	555-1234
67890	L.Van Pelt	34 Pear Ave.	555-5678
22222	P.Patty	56 Grape Blvd.	555-9999

(a) 学号–姓名–地址–电话

课　程	前　提
CS101	CS100
EE200	EE005
EE200	CS100
CS120	CS101
CS121	CS120
CS205	CS101
CS206	CS121
CS206	CS205

(b) 课程–前提

课　程	日　子	时　刻
CS101	M	9AM
CS101	W	9AM
CS101	F	9AM
EE200	Tu	10AM
EE200	W	1PM
EE200	Th	10AM

(c)课程–日子–时刻

课　程	教　室
CS101	Turing Aud.
EE200	25 Ohm Hall
PH100	Newton Lab

(d) 课程–教室

图8-2　样本关系

8.2.3 数据库的查询

我们在第7章中看到过对关系和函数执行的一些特别重要的操作，虽然根据处理的是词典、函数或是二元关系，它们相应的意义会有所区别，但这些操作都名为插入、删除和查找。我们能对数据库关系，特别是对两个或多个关系的结合，执行大量的操作，而且8.7节中还会概述对两个或多个关系的结合执行的操作。不过现在让我们将注意力集中在对单一关系执行的基本操作上。这些操作是对第7章中讨论过的那些操作的自然概括。

(1) *insert*(*t*,*R*)。如果元组*t*尚未出现在关系*R*中，就将它添加到*R*中。该操作与词典或二元关系的插入操作有着相同的精神。

(2) *delete*(*X*,*R*)。在这里，*X*是某些元组的规范。它是由对应*R*各属性的组分组成的，每个组分都会是下面两者之一。

(a) 一个值。

(b) 符号*，表示可以接受任意值。

该操作的效果是删除满足规范*X*的所有元组。例如，如果要取消课程CS101，就要从课程–日子–时刻关系中删除所有课程属性为“CS101”的元组。我们可以用

Lookup((“CS101”, *, 8)，课程 – 日子 – 时刻)

表示这种情况。该操作会删除图8-2(c)中的前3个元组，因为它们的第一个组分与该规范的第一个组分有着相同的值，而且它们的第二和第三个组分也都像任意值那样能与*匹配。

(3) *lookup*(*X*,*R*)。该操作的结果是得到*R*中匹配规范*X*的元组形成的集合，*X*是个象征性的元组，就跟第(2)项中描述的一样。例如，如果我们想要知道哪些课程是以CS101为前提，就可以询问

$$lookup\big((*,\text{"CS101"}),\text{课程}-\text{前提}\big)$$

结果是由两个与条件匹配的元组(CS120，CS101)和(CS205，CS101)组成的集合。

✦ 示例 8.2

下面有更多对教务数据库进行操作的例子。

(a) $lookup\big((\text{"CS101"},12345,*),\text{课程}-\text{学号}-\text{前提}\big)$ 可以找到学号为12345的学生CS101课程的成绩。正式地讲，得到的结果只有一个匹配的元组，也就是图8-1中的第一个元组。

(b) $lookup\big((\text{"CS205"},\text{"CS120"}),\text{课程}-\text{前提}\big)$ 会询问CS120是否为CS205的前提。正式地讲，如果元组 ("CS205","CS120") 在该关系中，产生的回答就是该元组，如果该元组不在该关系中，那么回答就是空集。对图8-2b中的关系而言，得到的回答是空集。

(c) $delete\big((\text{"CS101"},*),\text{课程}-\text{教室}\big)$ 会剔除图8-2d中的第一个元组。

(d) $insert\big((\text{"CS205"},\text{"CS120"}),\text{课程}-\text{前提}\big)$ 会使CS120成为CS205的前提。

(e) $insert\big((\text{"CS205"},\text{"CS101"}),\text{课程}-\text{前提}\big)$ 不会对图8-2b的关系造成影响，因为要插入的元组已经在该关系中了。

8.2.4 表示关系的数据结构的设计

在本章接下来的部分中，有大量篇幅用来讨论如何为关系选择数据结构的问题。在7.9节中

讨论二元关系的实现时，我们已经见识过有关该问题的一些内容。我们为品种–传粉者关系给定了一个有关品种的散列表作为其数据结构，而且我们看到该结构在回应诸如

$$lookup\big(("\text{Wickson}",*)，品种-传粉者\big)$$

这样的查询时会非常实用，因为值“Wickson”让我们找到了有待查找的特定散列表元。不过该结构对回应诸如

$$lookup\big((*,"\text{Wickson}")，品种-传粉者\big)$$

这样的查询是无所帮助的，因为我们必须要在所有的散列表元中查找。

有关品种的散列表是否为合适的数据结构，取决于预期的查询组合。如果我们预期品种总是已指定的，那么散列表就是合适的，而如果预期品种有时候是未指定的，就如先前的查询那样，那么就需要设计一种更强大的数据结构。

数据结构的选择是我们在本章中要面对的基本设计问题之一。在8.3节中，我们要推广7.8节和7.9节中用于函数和二元关系的基本数据结构，从而让一些属性要么在定义域中，要么在值域中。这些结构将被称为“主索引结构”。然后，在8.5节中要介绍“辅助索引结构”，它们是让我们能高效回应更多种类查询的额外的结构。到那时，我们就将看到如何能让上述两个查询以及其他与品种–传粉者关系有关的查询得到高效回应，也就是说，大约在列出所有这些回应所花的这段时间内。

设计I：数据库模式的选择

在使用关系数据模型时，如何选择合适的数据库模式是个重要的问题。例如，为什么我们要把与课程有关的信息分为5个关系，而不是将其放在具有以下模式的一张表中

{课程，学号，成绩，前提，日子，时刻，教室}

直觉上的原因在于下列两点。

❑ 如果将两个独立类型的信息结合成一个关系模式，就可能被迫多次重复同样的事实。

例如，与课程有关的前提信息，是独立于日子和时刻信息的。如果我们将前提信息与日子–时刻信息结合在一起，就不得不在列出某课程的前提时还要加上每次上课的时间，反之亦然。那么，如果将图8-2b和图8-2c中有关课程EE200的数据放在具有{课程，前提，日子，时刻}模式的单一关系中，就成了

课　程	前　提	日　子	时　刻
EE200	EE005	Tu	10AM
EE200	EE005	W	1PM
EE200	EE005	Th	10AM
EE200	CS100	Tu	10AM
EE200	CS100	W	1PM
EE200	CS100	Th	10AM

请注意，要完成之前各含2到3个组分的5个元组就能完成的工作，这里要用到6个元组，而且每个元组有4个组分。

❑ 反过来，在属性表示相互联系的信息时，不要把它们分开。

例如，我们不能把“课程–日子–时刻”关系替代为分别具有“课程–日子”模式和“课程–时刻”模式的两个关系。因为那样的话，我们只能告知EE200会在星期二、星期三和星期四上课，而且会在上午10点及下午1点上课，但没办法说明这3天分别是在几点上课。

8.2.5 习题

(1) 为图8-2a到图8-2d的这些关系中的元组给出合适的结构体声明。

(2) * 对以下问题来说，合适的数据库结构是什么?

(a) 电话簿，包含电话簿中所有常见信息，比如区号。

(b) 英语词典，包含词典中所有常见信息，比如词源和词性。

(c) 日历，包含日历中所有常见信息，比如节假日，要含有从公元1年到公元4000年的所有内容。

8.3 键

很多数据库关系可被视作从某些属性的集合到其余属性的函数。例如，我们可将“课程–学号–成绩”关系看作定义域为“课程–学号”有序对，值域为成绩的函数。因为函数的数据结构比一般关系的数据结构多少要简单一些，所以如果我们知道可以作为函数定义域的属性集合，是能派上用场的。这样的属性集合就叫作“键”。

更正式地讲，关系的键是一项或多项属性的集合，它满足这样的条件，就是在任何情况下，以键属性为标题的列中不会出现相同的值。通常，有很多不同的属性集合可以作为关系的键，不过我们一般只选择一个，并称为“键”。

8.3.1 键的确定

因为键可以用作函数的定义域，所以它们在8.4节中讨论主索引结构时扮演了重要角色。一般而言，我们没法证实或证明属性集合可以形成键，而是需要小心地检查与要建模的应用有关的假设，以及这些假设如何反映在我们设计的数据库模式中。只有这样才能知道给定的属性集合是否适合作为键。接下来有一系列的示例用来说明一些问题。

✦ 示例 8.3

考虑图8-2a中的“学号–姓名–地址–电话”关系。显然，每个元组都是用来表示不同学生的信息的。我们不希望找到两个具有相同学号的元组，因为这一编号存在的意义就是要为每个学生指定一个唯一的标识符。如果让同一个关系中有了两个学号相同的元组，那么其中有一个就是出错了。

(1) 如果两个元组所有的组分都相同，那么就违背了关系是集合的假设，因为集合中每个元素最多只能出现一次。

(2) 如果两个元组有着相同的学号，但在姓名、地址或电话这3列中至少有一个不同，就说明数据存在错误。要么是我们给了不同的学生同一个学号（如果元组中的“姓名”有区别），或是错给同一个学生记录了两个不同的地址和（或）电话号码。

因此，将“学号”属性作为“学号–姓名–地址–电话”关系的键是合理的。

不过，要将“学号”声明为键，就要作出一项关键的假设，就是在之前的第(2)项中阐明的，决不会为同一个学生存储两个姓名、地址或电话号码。不过我们还可能作出其他的决定，例如为每个学生存储家庭地址和校园地址。如果这样的话，就最好将该关系设计成具有5项属性，将“地址”属性替换为家庭地址（HomeAddress）和本地地址（LocalAddress）这两个属性，而不是为每个学生使用两个只有地址组分不同的元组，因为那样的话学号就不再能作为键了，而{学号，地址}就能作为键。

✦ 示例 8.4

审视图8-1中的“课程–学号–成绩”关系，我们可能想将“成绩”作为键，因为表中没有两个元组的成绩是相同的。不过，这种推理是不合理的。在这个只有6个元组的示例中，确实没有两个元组具有相同的成绩，不过在常见的“课程–学号–成绩”关系中，可能有着成千上万个元组，肯定有很多成绩会出现多次。

最有可能的是，数据库设计人员的想法是用“课程”和“学号”这两个属性一起形成键。也就是说，假设学生不可能选修同一课程两次，因此，不可能出现课程与学号都相同的两个不同元组。因为我们预见可以找出多个具有相同“课程”组分的元组，以及很多有着同样“学号”组分的元组，所以“课程”和“学号”都不能单独作为键。

不过，学生在任意课程中只可以获得一个成绩的假设也是有待推敲的，这取决于学校的具体政策。也许在课程内容发生重大改变时，学生可以重新注册该课程。如果有这种情况，就不能将{课程，学号}声明为“课程–学号–成绩”关系的键，只有全部3个属性的集合才可以作为键。请注意，具有关系中所有属性的集合总能作为键，因为关系中不可能出现两个相同的元组。事实上，最好是添加第四项属性，日期来表示课程被选择的时间，这样一来就可以处理学生选了同样课程两次并且两次都获得相同成绩的情况了。

✦ 示例 8.5

在图8-2b的“课程–前提”关系中，两项属性都不能单独作为键，不过两项属性一起就能形成键。

✦ 示例 8.6

在图8-2c所示的“课程–日子–时刻”关系中，全部3项属性一起形成了唯一合理的键。可能只要课程和日子加在一起就能声明为键，不过这样就没法存储同一天中要出现两次的课程了（比如演讲和实验）。

设计II：键的选择

为关系确定键是数据库设计中的一个重要方面，当我们在8.4节中选择主索引结构时就会用到。

- ❑ 不能只靠观察关系的几个示例值就确定键。

也就是说，外表可能有欺骗性，就像我们在示例8.4中讨论过的，图8-1所示“课程–学号–成绩”关系中的“成绩”属性那样。

- ❑ 不存在所谓的“正确的”键选择，选择什么属性作为键，取决于对关系所含数据的类型作出的假设。

✦ 示例 8.7

最后，考虑一下图8-2d中的“课程–教室”关系。我们认为“课程”可以作为键，也就是说，不会有课程会在两个或多个不同的教室中进行。如果情况不这样，就应该将“课程–教室”关系与“课程–日子–时刻”关系结合起来，这样就可以区分一门课程是在哪间教室里上课了。

8.3.2　习题

(1) * 假设我们想为“学号–姓名–地址–电话”关系中的学生分别存储家庭地址及本地地址，以及家庭电话及本地电话。

(a) 这样一来，该关系最为合适的键是什么？

(b) 这一改变会带来冗余，例如，如果某学生的两个地址和两个电话号码以所有可能的方式结合后出现在不同元组中，那么该学生的姓名就会重复4次。我们在示例8.3中提供过一种解决方案，就是使用不同的属性来表示不同的地址和不同的电话。那么这样一来关系模式会是怎样的？而这一关系最合适的键是什么？

(c) 8.2节中介绍过另一种处理冗余的方式，就是将该关系分解成两个具有不同模式的关系，一起存放原关系的所有信息。如果要为同一学生存储多个地址和电话，应该将“学号–姓名–地址–电话”关系分解为哪几个关系？提示：关键问题在于地址和电话是否为独立的。也就是说，是否期望一个电话号码会在某学生的所有地址都能接通（在地址和电话相互独立的情况下），还是说电话号码是与单个地址相关联的。

(2) * 车管所维护着含有如下若干类信息的数据库。

- ❑ 驾驶员的姓名（Name）。
- ❑ 驾驶员的地址（Addr）。
- ❑ 驾驶员的驾驶证编号（LicenseNo）。
- ❑ 车辆的序列号（SerialNo）。
- ❑ 车辆的生产商（Manf）。
- ❑ 车辆的型号名称（Model）。
- ❑ 车辆的注册（车牌）号（RegNo）。

车管所希望将每个驾驶员关联到相关信息：地址、驾驶证和所拥有的车辆。还希望将每辆车关联到相关信息：所有者、序列号、生产商、型号和注册号。我们假设熟悉车管所运作的基本要求，例如，不可能将同样的车牌发给两辆汽车。大家可能不知道（但这确实是事实），即便是来自不同生产商的两辆汽车，也不可能具有同样的序列号。

(a) 选择数据库模式，也就是关系模式的集合，其中每个关系模式都由上面列出的从1到7这几种属性的集合组成。大家必须让所需的联系都可以通过存储在这些关系中的数据体现出来，而且必须避免冗余，也就是说，大家设计的模式不应该重复存储相同的事实。

(b) 说明哪些属性可以作为(a)小题中设计的关系的键。

8.4　关系的主要存储结构

在7.8节和7.9节中，我们看到如何通过根据有序对的定义域值存储有序对，以使对函数和二元关系的某些操作提速。在提到我们在8.2节中定义的一般的插入、删除和查找操作时，能有所帮助的是那些指定了定义域值的操作。再次回想一下7.9节中的“品种–传粉者”关系，如果将品种作为关系的定义域，就有利于那些指定了品种而不关心是否指定了传粉者的操作。

这里有一些可用于表示关系的结构。

(1) 二叉查找树，在定义域值上有“小于”关系以安排元组的位置，可以用来促进指定了定义域值的操作。

(2) 以定义域值作为数组索引，用作特征向量的数组有时是有用的。

(3) 散列定义域值以找到散列表元的散列表是有用的。

(4) 原则上讲，元组组成的链表是一种候选结构。我们将忽略这种可能，因为它对任何类型的操作都没有促进作用。

当关系不是二元关系时，同样的结构也是可以使用的。定义域不是只有单个属性，而是可能结合k个属性，我们将其称为*定义域属性*，或是在明确所指的是属性集合时，将其直接称为“定义域”。这样一来，定义域的值就是k元组，各组分对应定义域的各属性。而*值域属性*是那些定义域属性之外的属性。值域值也可以有多个组分，每一个都对应着一个值域的属性。

一般来说，我们必须选出想要作为定义域的那些属性。最简单的情况是一个属性或少量属性作为关系的键的情况。这样的话就可以选择键属性作为定义域，而将其余属性作为值域。在没有键的情况下（所有属性的集合这种不实用的键除外），我们可以选择任意属性集合作为定义域。例如，可以考虑期望对该关系执行的那些常用操作，并选择预期经常要指定的属性作为定义域。我们很快就将看到一些具体的例子。

一旦选择了定义域，就可以从刚提到的4种数据结构中任选其一表示该关系，或者其实也可以选择另一种结构。不过，通常会选择以定义域值作为索引的散列表，而且我们在这里一般都会这么做。

所选的结构就称为该关系的*主索引结构*。形容词“主”表示元组的位置是由该结构确定的。*索引*则是在给定所需要的元组的一个或多个组分的情况下协助找到元组的数据结构。在8.5节中，我们将讨论“辅助”索引，它有助于回应查询，但不影响数据的位置。

```
typedef struct TUPLE *TUPLELIST;
struct TUPLE {
    int StudentId;
    char Name[30];
    char Address[50];
    char Phone[8];
    TUPLELIST next;
};
typedef TUPLELIST HASHTABLE[1009];
```

图8-3　作为主索引结构的散列表的类型

✦ 示例 8.8

我们来考虑一下以“学号”属性作为键的“学号–姓名–地址–电话”关系。学号属性就将作为定义域，而其他3个属性则会形成值域，因此可以将该关系视为从学号到“姓名–地址–电话”三元组的函数。

就和所有的函数那样，我们选择接受定义域值作为参数并生成散列表元号作为结果的散列函数。在这种情况下，散列函数会接受学生的学号（整数）作为参数。我们将选择1009[①]作为散列表元的数量B，这样散列函数就是

$$h(x) = x\%1009$$

散列函数将学号映射到从0到1008这个范围的整数。

含1009个散列表元头部的数组给我们带来了一列列的结构体。i号散列表元对应的链表中的结构体表示的是学号组分除以1009余数为i的那些元组。对“学号–姓名–地址–电话”关系来说，

① 1009是1000左右的质数。如果数据库要记录数千学生的信息，就可以使用1000个左右的散列表元，这样一来每个散列表元中的平均元组数就会很小。

图8-3中的声明对散列表元链表中的结构体和散列表元头部数组来说都是合适的。图8-4展示了散列表看起来的样子。

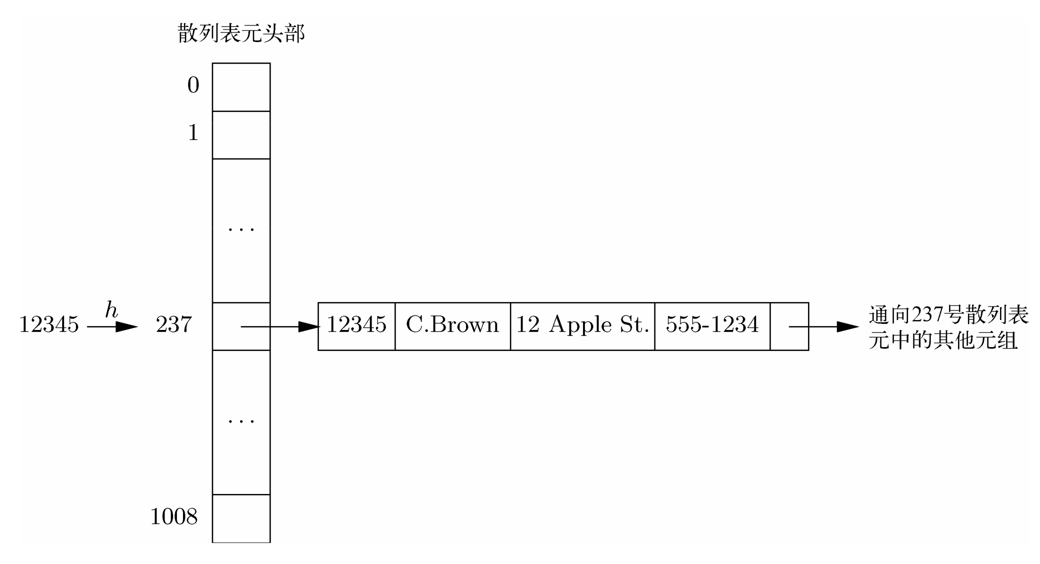


图8-4 表示“学号–姓名–地址–电话”关系的散列表

✦ 示例 8.9

再看一个更复杂的例子，考虑一下“课程–学号–成绩”关系。我们可以用散列表作为主结构，该散列表的散列函数接受课程和学号（即构成该关系的键的两项属性）作为参数。这样的散列函数要接受表示课程名称的字符，再加上表示学生学号的整数，然后再除以1009取余数。

如果我们要进行的操作都是在给定课程与学号的情况下查找成绩，这一数据结构就很实用。也就是说，它适用于执行如下操作：

$$lookup((\text{“CS101”}, 12345,*),\text{课程}-\text{学号}-\text{成绩})$$

不过它对诸如下面这样的操作来说就不实用。

(1) 找出所有选修CS101课程的学生；

(2) 找出学号12345的学生选修的所有课程。

在这两种情况下，我们都没法计算散列值。例如，只给定课程，就没有学号可加到课程名字符对应整数的和上，因此就没有值除以1009以得出散列表元号。

不过，假设经常要进行“谁选修了CS101”这样的查询，也就是

$$lookup((\text{“CS101”}, *, *),\text{课程}-\text{学号}-\text{成绩})$$

那么使用只基于“课程”组分的值的主结构会更有效。也就是说，可以将该关系视作集合论中的二元关系，其中定义域是课程，而值域则是学号–成绩有序对。

例如，假设我们将课程名称的字符转换成整数，并求出它们的和，除以197，然后取余数。那么“课程–学号–成绩”关系的元组就会被该散列函数分装到标号为0至196的197个散列表元中。不过，如果有100个学生选修了CS101课程，那么不管我们为散列表安排了多少个散列表元，对应课程CS101的散列表元中都至少有100个结构体，这就是使用不是键的属性作为主索引结构的定义域的缺点。如果其他课程也被散列到对应CS101的散列表元，那么该散列表元中甚至会有

100个以上的结构体。

另一方面，当想要在给定的课程中找到学生时，仍然能从中受益。如果课程的数量明显大于197，那么平均下来，只需要查找整个“课程–学号–成绩”关系的1/197，这是一笔巨大的节省。此外，当我们执行在特定课程中查找特定学生的成绩，或是插入或删除“课程–学号–成绩”元组这样的操作时，也会受益于该结构。在每种情况下，都可以使用“课程”值将查找范围限定在散列表197个散列表元中的某一个上。唯一帮不上忙的情况就是处理没有指定课程的操作。例如，要找到学生12345选修的课程，就必须查找所有的散列表元。这样的查询只有在使用辅助索引结构时才可以更有效率地进行，这一点会在8.5节中讨论。

设计III：主索引的选择

- 将关系模式的键作为函数的定义域，并将其余属性作为值域通常是很实用的。

然后就可以像实现函数那样，使用诸如带有基于键属性的散列函数的散列表这样的主索引来实现关系。

- 不过，如果最常见的查询所指定的是不构成键的属性的值，就可能要选用该属性集合作为定义域，而将其余属性作为值域。

  接着就可以像实现二元关系那样来实现该关系了。比如，利用散列表。唯一的问题就在于，元组在散列表元中的分布可能不像以键作为定义域时那么平均。
- 主索引结构定义域的选择可能对执行“常规”查询的速度有着最大的影响。

8.4.1　插入、删除和查找操作

鉴于第7章中对二元关系的同样主题的探讨，这里用主索引结构执行插入、删除和查找操作的方式应该很明了。要回顾这些概念，就要将注意力放在作为主索引结构的散列表上。如果操作指定了定义域的值，那么就要散列该值以找到散列表元。

(1) 要插入元组t，就要检查相应的散列表元，看看t是否已经位列其中，如果没有就在该散列表元对应的链表中创建新单元来容纳t。

(2) 要删除匹配规范X的元组，就要根据X找出定义域值，进行散列以得出相应的散列表元，然后沿着该散列表元对应的链表向下查找，将匹配规范X的各元组都删除掉。

(3) 要根据规范X查找元组，还是要从X找到定义域值，进行散列以得出相应的散列表元。沿着对应该散列表元的链表向下查找，将链表中匹配规范X的各元组分别作为回应生成。

如果操作没有指定定义域值，就不会这么走运了。插入操作就总是要完整地指定被插入的元组，而删除或查找操作可能不能这样。在那样的情况下，我们必须对所有的散列表元列表进行查找，找到匹配的元组，并分别删除或列出它们。

8.4.2　习题

(1) 8.3节中习题(2)的车管所数据库应该设计成能处理如下类型的查询，而且要假设这些查询发生的频率都相当高。

(a) 给定驾驶员的地址是什么？

(b) 给定驾驶员的驾驶证编号是多少？

(c) 给定驾驶证编号对应驾驶员的姓名是什么？
(d) 拥有给定车辆（以其注册号作为标识）的驾驶员的姓名是什么？
(e) 具有给定注册号的车辆的序列号、生产商和型号各是什么？
(f) 拥有给定注册号所对应车辆的是谁？

为自己在8.3节习题(2)中设计的那些关系给出合适的主索引结构，每种情况下都使用散列表。陈述有关驾驶员数与车辆数的假设。说出要使用多少散列表元，以及要使用什么属性作为定义域。这几类查询中有多少可以得到高效的回应，也就是说，平均只花费 $O(1)$ 的时间而不用考虑关系的大小。

(2) 图8-2c中“课程–日子–时刻”关系的主结构可能取决于我们打算执行的常见操作。如果常需要执行的操作分别为下面列出的这几类，给出合适的散列表，要说清定义域中的属性以及散列表元的数量。大家可以对课程的数量以及不同的上课时段作出合理假设。在每种情况下，像“CS101”这样的指定值是用来表示“常见”值的，这样的话，我们的意思是“课程”被指定为某一特定的课程。

(a) $lookup\big(("CS101","M",*), 课程-日子-时刻\big)$。
(b) $lookup\big((*,"M",*,"9AM"), 课程-日子-时刻\big)$。
(c) $lookup\big(("CS101",*,*), 课程-日子-时刻\big)$。
(d) (a)类和(b)类各占一半。
(e) (a)类和(c)类各占一半。
(f) (b)类和(c)类各占一半。

8.5　辅助索引结构

假设把“学号–姓名–地址–电话”关系存储到如图8-4所示、散列函数是基于“学号”键的散列表中。该主索引结构有助于回应那些指定了学生学号的查询。不过，我们可能希望以学生的姓名提问，而不是用客观而且可能未知的学号提问。例如，我们可能会问，“名叫C.Brown的学生的电话号码是多少？”这样一来主索引结构就帮不上忙了。我们必须行经每个散列表元，并检查一列列的记录，直到找到“姓名”字段的值为“C.Brown”的记录为止。

要迅速回应这样的查询，就需要额外的数据结构让我们可以用姓名找到“姓名”组分中含有该姓名的元组[①]。可以在给定某一属性或某些属性的值的情况下帮我们找到元组，但不能用来在整个结构中放置元组的数据结构，就是**辅助索引**。

这里我们需要的辅助索引是具备以下两个条件的二元关系。

(1) 定义域是“姓名”。

(2) 值域是指向“学号–姓名–地址–电话”关系的元组的指针。

一般而言，关系R属性A上的辅助索引是满足以下条件的有序对(v, p)的集合。

(a) v是属性A的值。

(b) p是指向关系R主索引结构中某个元组的指针，该元组的“A”组分的值为v。

对属性A的值为v的各元组来说，辅助索引都有对应的有序对。

① 要记住，“姓名”并不是“学号–姓名–地址–电话”关系的键，尽管在图8-2a所示的样本关系中，各元组的姓名组分的值都是不同的。例如，如果Linus和Lucy上了同一所大学，那么就有两个元组的姓名组分等于“L. Van Pelt”，但这两个元组的学号组分是不同的。

我们可以使用表示二元关系的任意数据结构来存储辅助索引。通常会期望使用基于属性A的值的散列表。只要散列表元的数量不大于属性A不同值的数量，在给定所需的v值的情况下，在散列表中查找有序对(v, p)通常都可以预期不错的性能——也就是平均$O(n / B)$的时间。这里的n是有序对的数量，而B是散列表元的数量。为了表明其他的结构也可以用于辅助索引（或主索引），我们在下一个示例中要使用二叉查找树作为辅助索引。

✦ 示例 8.10

我们来为图8-2a所示的“学号–姓名–地址–电话”关系设计数据结构，其中使用基于学号的散列表作为主索引，而使用二叉查找树作为对应姓名属性的辅助索引。为了简化表达，这里要使用只含两个散列表元的散列表作为主结构，而要使用的散列函数是用学号除以2的余数。也就是说，偶数学号会放进0号散列表元，而奇数学号会放进1号散列表元。

```
typedef struct TUPLE *TUPLELIST;
struct TUPLE {
    int StudentId;
    char Name[30];
    char Address[50];
    char Phone[8];
    TUPLELIST next;
};

typedef TUPLELIST HASHTABLE[2];

typedef struct NODE *TREE;
struct NODE {
    char Name[30];
    TUPLELIST toTuple; /* 其实是指向元组的指针 */
    TREE lc;
    TREE rc;
};
```

图8-5　对应主索引和辅助索引的类型

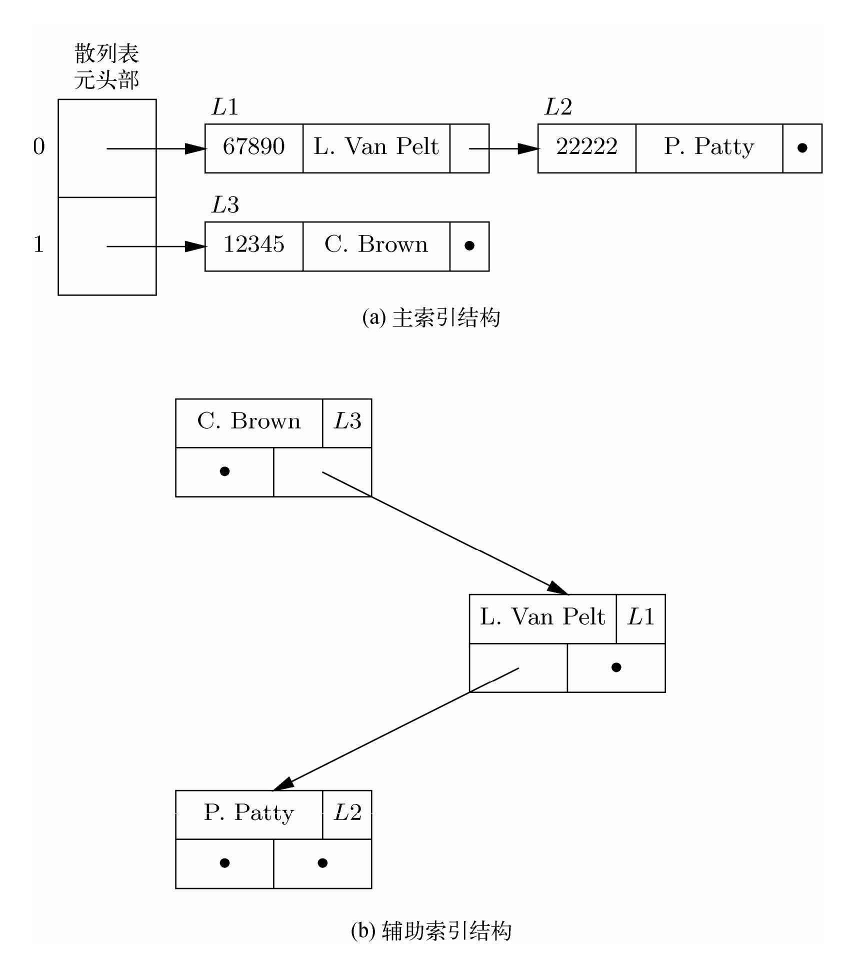
这里将使用二叉查找树作为辅助索引，该二叉查找树的节点中存储着由学生姓名与指向元组的指针组成的有序对。元组本身被存储为记录，这些记录链接成链表，构成了散列表的散列表元，所以指向元组的指针实际上就是指向记录的指针。因此，我们需要如图8-5所示的结构体。`TUPLE`和`HASHTABLE`类型与图8-3中的定义是一致的，只不过现在使用的是两个散列表元而不是1009个。

`NODE`类型是二叉树的节点，它具有`Name`和`toTuple`这两个字段，分别表示该节点处的元素（即某一学生的姓名），以及指向该学生对应的元组所在记录的指针。而其余的两个字段，`lc`和`rc`，分别是指向该节点左子节点与右子节点的指针。我们会用学生的姓的字母表次序作为比较树中接点处各元素所使用的“小于”次序。而辅助索引本身是`TREE`类型的变量，也就是指向节点的指针，它会将我们带到该二叉查找树的根。

图8-6展示了整个结构体的一个示例。为了节省空间，元组的地址和电话组分并没有表示出来。而图中的$L1$和$L2$等字样表示主索引结构中的记录在内存中的存储位置。

现在，如果想要回应诸如“P.Patty的电话号码是多少”这样的查询，就要从辅助索引的根

开始，查找Name字段为“P.Patty”的节点，并跟着该指针到达toTuple字段，如图8-6中的$L2$所示。这样就可以找到P.Patty的记录，并从该记录查阅Phone字段并生成该查询的回应。



(a) 主索引结构

(b) 辅助索引结构

图8-6　主索引结构与辅助索引结构的示例

8.5.1　非键字段上的辅助索引

看起来在示例8.10中构建辅助索引所依据的“姓名”属性是键，因为没有重复出现的姓名。不过，正如我们所知，存在两个学生同名的可能，所以“姓名”其实不是键。正如在7.9节讨论过的，虽然非键属性可能让元组在散列表元中的分布不如预期的那样平均，但它并不会影响到散列表的数据结构。

二叉查找树是另一个问题，因为这种数据结构不能处理不存在“小于”关系的两个元素，如果两个有序对有着相同的姓名和不同的指针，就会出现这种情况。对图8-5所示结构进行一个小的修正，用字段toTuple作为指向元组的指针组成的链表的表头，具有Name字段中给定值的各元组都与一个指针对应。例如，如果有很多个P.Patty，那么图8-6b中底部的节点就会在$L2$的位置具有链表的表头。而该链表中的元素就是指向“姓名”属性等于“P.Patty”的各元组的指针。

设计VI：何时应该创建辅助索引？

如果元组的一个或多个组分的值已经给定，辅助索引的存在通常会让查找元组的工作变得更容易。不过还要考虑如下两点。

- 所创建的每个辅助索引都会让我们在关系中插入或删除信息时花费额外的时间。
- 因此，只为那些可能需要查找数据的属性构建辅助索引是说得通的。

例如，如果从不打算在只给定电话号码的情况下找到学生，就没必要在“学号–姓名–地址–电话”关系中的“电话”属性上创建辅助索引。

8.5.2　辅助索引结构的更新

当某个关系存在辅助索引时，元组的插入和删除操作就会变得更困难。除了要像8.4节概述的那样更新主索引结构，还可能需要更新各辅助索引结构。以下方法可用来在涉及属性A的元组被插入或删除时更新基于A的辅助索引结构。

(1) 插入。如果要插入一个新元组，其对应属性A的组分的值为v，就必须创建有序对(v,p)，其中p是指向主结构中新记录的指针。然后，再把有序对(v,p)插入到辅助索引中。

(2) 删除。要删除对应A的组分的值为v的元组时，首先一定要记得已经删除了指向该元组的指针，比方说是p。然后，要深入辅助索引结构，并检查所有第一个组分为v的有序对，直到从其中找出第二个组分为p的有序对为止。然后将该有序对从辅助索引结构中删除。

8.5.3　习题

(1) 给出如何修改图8-5中的二叉查找树结构，以使“学号–姓名–地址–电话”关系可以存在学生姓名相同的多个元组。编写C语言函数，接受姓名作为参数，并列出关系中“姓名”属性为该姓名的所有元组。

(2) ** 假设已决定用“学号”属性上的主索引来存储“学号–姓名–地址–电话”关系，还决定创建一些辅助索引。假设所有的查找都只会指定姓名、地址或电话属性中的某一个。并假设所有的查找操作中有75%是指定了姓名的，有20%是指定地址的，还有5%是指定电话的。还假设每次插入或删除操作的开销都是1个时间单位，再加上我们构建的每个辅助索引会用掉1/2个时间单位。比方说，如果我们要构建所有3个辅助索引的话，总的时间开销就是2.5个时间单位。设如果指定了具有辅助索引的属性，那么一次查找的开销是1个时间单位，而如果指定的属性没有辅助索引则会花费10个时间单位。设a是对指定了全部3项属性的元组进行的插入和删除操作所占的比例。其余的$1-a$是指定了某一项属性的操作所占比例，并且符合我们之前对这类查找操作出现概率的假设。比如，所有操作中有$0.75(1-a)$的比例是给定了“姓名”值的查找。如果目标是让一次操作的平均时间最小化，那么当参数a的值分别为(a)0.01；(b)0.1；(c)0.5；(d)0.9；(e)0.99时，分别应该创建哪些辅助索引？

(3) 假设车管所希望能高效回应如下类型的查询，也就是说，要比查找整个关系快得多。

(i) 给定驾驶员的姓名，找到发放给具有该姓名的人们的驾驶证。

(ii) 给定驾驶证编号，找到驾驶员的姓名。

(iii) 给定驾驶证编号，找到该驾驶员拥有的车辆的注册号。

(vi) 给定地址，找到所有登记为该地址的驾驶员的姓名。

(v) 给定注册号（即车牌号），找到该车辆所有者的驾驶证。

为8.3节习题(2)中建立的关系给出合适的数据结构，从而使得这些查询能得到高效回应。假设每个

索引都是从散列表构建的，并说出各关系的主索引结构和辅助索引结构，这样就足够了。解释一下这样一来要如何回应各类型的查询。

(4) * 假设需要高效地从给定的辅助索引中找到指向主索引结构中特定元组t的指针。给出数据结构，让我们能在与找到的指针数成正比的时间内找到这些指针。哪些操作会因为这一额外的数据结构而变得更加费时？

8.6 关系间的导航

直到现在，我们只考虑了涉及单一关系的操作，比如在给定元组的一个或多个组分的值的情况下找到该元组。关系模型的威力要得到最好的体现，就需要考虑那些要求我们“导航”，或者说从一个关系跳转到另一个关系的操作。例如，我们在回应“学号为12345的学生CS101课程的成绩是多少”这样的查询时，所有的处理都是在“课程–学号–成绩”关系之内展开的。不过，如果查询是更为自然的“C.Brown在CS101课程中取得了怎样的成绩”呢？该查询只在“课程–学号–成绩”关系之内就不能得到回应了，因为该关系使用的是学号，而非姓名。

要回应这一查询，首先必须查阅“学号–姓名–地址–电话”关系，并将姓名C.Brown转换成一个学号，或若干学号，因为有可能存在两个或多个学生姓名相同而学号不同的情况。然后，对每个这样的学号，都要在“课程–学号–成绩”关系中查找对应该学号而且课程组分为CS101的元组。可以从每个这样的元组中读取出名为C.Brown的学生CS101课程的成绩。图8-7表示了该查询是如何将给定值与这些关系以及所需的回应联系在一起的。

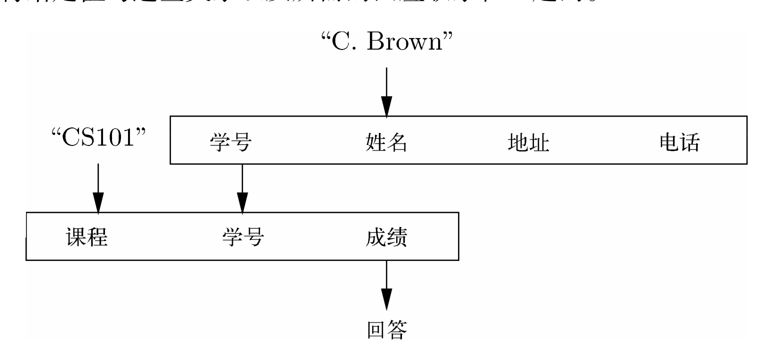


图8-7 表示查询“C.Brown在CS101课程中取得了怎样的成绩”的图

如果没有索引可用，回应该查询就可能会相当费时。假设在“学号–姓名–地址–电话”关系中有n个元组，而且在“课程–学号–成绩”关系中有m个元组。再假设名叫C.Brown的学生共有k个。在假设没有索引可用的情况下，找出这一（或这些）学生在CS101课程拿到的成绩的算法提纲就如图8-8所示。

接着来确定图8-8所示程序的运行时间。从里层开始向外分析，第(6)行的打印语句要花$O(1)$的时间。而第(5)和第(6)行的条件语句也要花$O(1)$的时间，因为第(5)行的测试是$O(1)$时间的测试。由于我们假设在“课程–学号–成绩”关系中有m个元组，这样一来，第(4)到第(6)行的循环要迭代m次，因此总共要花$O(m)$的时间。因为第(3)行花费的是$O(1)$时间，所以第(3)到第(6)行的程序块花的时间就是$O(m)$。

现在考虑一下第(2)到第(6)行的`if`语句。因为第(2)行的测试花费$O(1)$的时间，所以如果条件为假则整个`if`语句花费$O(1)$时间，如果为真则花费$O(m)$的时间。不过，我们已经假设了该

条件对k个元组为真而对其余元组为假，也就是说，有k个元组t的姓名组分是C.Brown。因为当条件为真与条件为假时所花的时间存在很大的区别，所以应该对分析第(1)到第(6)行的`for`循环的方式格外谨慎小心。也就是说，我们不能记下循环迭代的次数并将其乘上循环体可能花费的最大时间，而是要分开考虑第(2)行测试的两种结果。

```
(1)  for "学号-姓名-地址-电话"关系中的各元组 t do
(2)      if  t 的"姓名"组分为"C. Brown"  begin
(3)          设i 是元组t的"学号"组分;
(4)          for "课程-学号-成绩关系"中的各元组s do
(5)              if s 的"课程"组分为"CS101"且
                   "学号"组分为i then
(6)                      print 元组s的"成绩"组分;
         end
```

图8-8　找到C.Brown在CS101课程取得的成绩

首先，要进行n次循环，因为这是不同t值的数量。对令第(2)行的测试为真的k个元组t来说，我们在每个元组上所花时间为$O(m)$，或者说总共要花上$O(km)$的时间。对其余$n-k$个令该测试为假的元组来说，每个元组要花$O(1)$的时间，或者说总共要花$O(n-k)$的时间。因为估计k是大大小于n的，所以我们选择$O(n)$而非$O(n-k)$作为更简单的紧上界。因此整个程序的时间开销就是$O(n+km)$。在很可能出现的$k=1$的情况中，当只有一个学生姓名为C.Brown时，所需的时间就是$O(n+m)$，它是与所涉及两个关系的大小之和成正比的。如果k大于1，这个时间就会更大。

8.6.1　利用索引为导航提速

有了合适的索引，我们在回应同样的查询时平均只需要$O(k)$的时间，也就是说，如果名字叫C.Brown的学生数k为1，就只需要$O(1)$的时间。这样是说得通的，因为我们肯定会检查$2k$个元组，就是两个关系各要检查k个。如果散列表使用了数量恰当的散列表元，这些索引让我们能以平均每个元组$O(1)$时间的开销找到所需的元组。如果拥有对应“学号–姓名–地址–电话”关系的“姓名”索引，以及对应“课程–学号–成绩”关系的“课程–学号”有序对上的索引，那么找出C.Brown在CS101课程中取得的成绩的算法可以大致描述为图8-9所示的样子。

```
(1)  使用"姓名"上的索引，找到"学号-姓名-地址-成绩"关系中
     "姓名"组分为"C. Brown"的各元组；
(2)  for 步骤(1)中找到的各元组t do begin
(3)  设i 是元组t 的"学号"组分；
(4)      使用"课程-学号-成绩"关系中"课程"和"学号"上的索引，
          找到"课程"组分为"CS101"而且"学号"组分为i的元组s；
(5)  print 元组s的"成绩"组分；
    end
```

图8-9　使用索引找到C.Brown在CS101课程中取得的成绩

我们假设“姓名”索引是具有约n个散列表元的散列表，是用作辅助索引的。因为n是“学号–姓名–地址–成绩”关系中的元组数量，所以每个散列表元中平均有$O(1)$个元组。如果具有该姓名的元组有k个，在散列表元中找到这些元组就要花费$O(k)$的时间，而且跳过该散列表元中可能存在的其他元组要花$O(1)$的时间。因此，图8-9的第(1)行平均要花$O(k)$的时间。

第(2)到第(5)行的循环要执行k次。假设把在第(1)行找到的k个元组t存储在一个链表中，那么不管是找到下一个元组t，还是发现没有更多的元组，进行循环所花的时间都为$O(1)$，而且第(3)到第(5)行的开销也是一样的。我们声明第(4)行也能在$O(1)$时间内执行，因此第(2)到第(5)行的运行时间也是$O(k)$。

下面来分析一下第(4)行。第(4)行要求在给定某一元组的键值的情况下对其进行查找。假设“课程–学号–成绩”关系具有其键{课程，学号}上的主索引，而且该索引是约有m个散列表元的散列表。那么，每个散列表元中所含元组的平均数就是$O(1)$，因此图8-9的第(4)行所花的时间就是$O(1)$。这样我们就能得出第(2)到第(5)行的循环体平均会花费$O(1)$的时间，因此图8-9所示的整个程序就平均会花费$O(k)$的时间。也就是说，这一时间开销是与具有我们要查询的姓名的学生的数量成比例的，而不用考虑涉及的关系的大小。

8.6.2 多关系上的导航

有些导航技巧让我们可以高效地从一个关系跳转到另一个关系，这些技巧也可以用于涉及多个关系的导航。例如，假设想知道“C.Brown星期一上午9点在哪里？”假设他在上课，我们就可以通过如下方式回应该查询，找到C.Brown选修的课程，看看其中是否有课是在星期一上午9点上，如果有的话，找到该课程上课的教室就行了。图8-10展示了从给定值C.Brown到得出的回应期间在各关系间的导航。

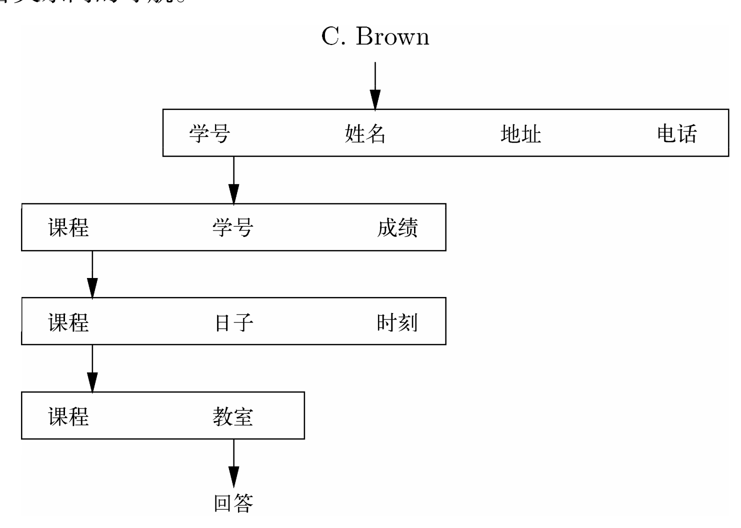


图8-10 表示“C.Brown星期一上午9点在哪里”这一查询的图

以下方案假设只有一个名为C.Brown的学生，如果有多个名叫C.Brown的学生，就可以得到星期一早上9点他们中的一个或多个人所在的教室。这里还假设这名学生没有选修相互冲突的课程，也就是说，他在星期一早上9点最多只上一门课。

(1) 利用对应C.Brown的“学号–姓名–地址–电话”关系，找到C.Brown的学号，设他的学号为i。

(2) 在“课程–学号–成绩”关系中查找所有学号组分为i的元组，设$\{c_1, \cdots, c_k\}$是这些元组中“课程”值的集合。

(3) 在“课程–日子–时刻”关系中，查找“课程”组分为c_i（即第(2)步中找到的课程之一）

的元组。应该最多只有一个元组的“日子”组分为“M”而且“时刻”组分为“9AM”。

(4) 如果第(3)步中找到的是课程c，那么在“课程–教室”关系中查找c上课的教室。假设C.Brown没打算旷课，这就是他星期一上午9点所在的位置。

如果没有索引，那么我们可以期待的最佳情况就是，在与涉及的4个关系的大小之和成比例的时间内完成这一方案。不过，有若干个索引可供我们利用。

(a) 在第(1)步中，可以使用“学号–姓名–地址–电话”关系中的“姓名”组分作为索引，从而在平均为$O(1)$的时间内得到C.Brown的学号。

(b) 在第(2)步中，假设C.Brown选修了k门课程，可以利用“课程–学号–成绩”关系中的“学号”组分上的索引，在$O(k)$的时间内得到C.Brown选修的所有课程。

(c) 在第(3)步中，可以利用“课程–日子–时刻”关系中“课程”组分上的索引，这样一来，就可以在与第(2)步得到的k门课程每周上课次数之和成比例的时间内，找出这些课程全部的上课时间。如果假设每门课程每周上课次数不超过5次，那么最多只有$5k$个元组，因此可以在$O(k)$的平均时间内找到它们。如果没有该关系“课程”属性上的索引，而是有“日子”和（或）“时刻”属性上的索引，虽然可能要查看远多于$O(k)$数量的元组（取决于星期一要上多少课，或是某一天的9点要上多少课），但还是能从这种索引中受益。

(d) 在第(4)步中，可以利用“课程–教室”关系中“课程”属性上的索引。在这种情况下，可以在平均为$O(1)$的时间内检索到所要找的教室。

这样就可以得出这样的结论，有了所有这些合适的索引，可以在$O(k)$的平均时间内回复这些非常复杂的查询。因为可以假设C.Brown所选课程的数量k很小，比方说5门左右，那么这一时间通常会特别少，而且特别要指出的是，该时间与所涉及关系的大小都无关。

总结：关系的快速访问

回顾一下我们从渐趋复杂的关系中获得答案的方式是很实用的。首先是在7.8节使用散列表或诸如二叉查找树或（概括化的）特征向量这样的结构实现函数，按照本章中的内容来看就是定义域为键的二元关系。然后，在7.9节中看到，只要关系是二元的，这些概念也适用于定义域不为键的情况。

在8.4节中看到，并不需要要求关系是二元的，可以将属于键的一部分的全部属性作为一个“定义域”集合，而将所有其他属性作为一个“值域”集合。此外，我们在8.4节中还看到定义域不一定要是键。

在8.5节中我们了解到，可以使用某关系上的多个索引结构，提供基于不属于定义域的属性的快速访问。而且在8.6节中我们看到，可以结合多个关系上的索引，在与实际查阅的元组数成比例的时间内执行复杂的信息检索。

8.6.3　习题

(1) 假设图8-9中的“课程–学号–成绩”关系不具备“课程–学号”对上的索引，而是只有课程这一个属性上的索引。这会对图8-9所示程序的运行时间造成怎样的影响？如果索引只建立在“学号”属性上呢？

(2) 讨论一下如何高效地回应下列查询。在每种情况下都要陈述对中间集合的元素数量（比如，C.Brown所选课程的数量）作出的假设，还要讲出都假设有哪些索引存在。

(a) 找到C.Brown所选课程的所有前提。

(b) 找到会在Turing Aud.上课的所有学生的电话号码。

(c) 找到CS206课程的前提课程的前提。

(3) 假设没有索引，那么习题(3)中的各查询要花多少时间，将其表示为所涉及关系的大小的函数，其中要对所有元组进行迭代，就像本节的那些示例中一样。

8.7 关系代数

我们在8.6节中看到，涉及多个关系的查询可能是相当复杂的。用一种比C语言“高级得多”的语言来表示这样的查询是很实用的，和C语言不同的是，这种语言中的查询在表示我们想要的内容时，比如，“课程”组分等于CS101的所有元组，可以不需要处理诸如在索引中进行查找操作这样的问题。出于这种目的，一种名为*关系代数*的语言应运而生。

就像任何代数那样，关系代数让我们可以应用代数法则改写查询。因为复杂的查询通常有很多不同的步骤序列，我们要借助这些步骤从存储的数据中得到查询的回应，而且因为不同的步骤序列是由不同的代数表达式表示的，所以关系代数提供了一个将代数作为设计理论的绝佳例子。其实，这种借助关系代数表达式的变形实现的效率提升，可视作代数的力量在计算机科学中得到体现的最突出示例。代数变形带来的“优化”查询的能力是8.9节的主题。

8.7.1 关系代数的操作数

在关系代数中，操作数都是关系。与其他代数一样，这里的操作数既可以是常量——在这种情况下就是指定的关系，也可以是表示未知关系的变量。不过，不管是变量还是常量，每个操作数都有特定的模式（为关系中的列命名的属性的集合）。因此，常量参数可能是像下面这样

A	*B*	*C*
0	1	2
0	3	4
5	2	3

该关系的模式是$\{A，B，C\}$，它含有3个元组，(0，1，2)、(0，3，4)和(5，2，3)。

变量参数可以用$R(A,BC)$表示，它表示被称为R的关系，该关系的各列分别名为A、B和C，但其组分集合是未知的。如果关系R的模式$\{A,B,C\}$是可以理解或是无关紧要的，就可以将R当作操作数。

8.7.2 关系代数的集合运算符

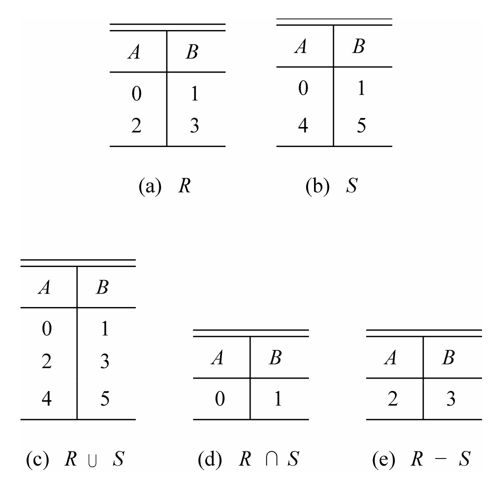
首先要使用的3种运算符是常见的集合元算：并、交、差，我们在7.3节中讨论过这些运算。这里要对这些运算符的操作数提出一个要求：两个操作数的模式一定要相同。这样一来，结果的模式就自然是这两个参数的模式。

✦ 示例 8.11

设R和S分别是图8-11a和图8-11b中的关系。请注意，这两个关系的模式都是$\{A,B\}$。并集运算符会生成这样一个关系，其中各元组要么在R中，要么在S中，或者是在两者之中。请注意，

因为关系是集合，所以即便某元组可能同时出现在R和S中，该元组在得到的关系中也最多只能出现一次，就像本例中的元组(0,1)这样。关系$U \cup S$如图8-11c所示。

交集运算符会生成由同时出现在两个操作数中的元组构成的关系。因此，关系$R \cap S$只含有元组(0,1)，如图8-11d所示。差集运算生成的关系包含那些在第一个关系中而不在第二个关系中的元组。关系$R - S$如图8-11e所示，具有R中的元组(2,3)，因为该元组不在S中，而它不含R中的元组(0,1)，因为这一元组也在S中。



A	B
0	1
2	3

(a) R

A	B
0	1
4	5

(b) S

A	B
0	1
2	3
4	5

(c) $R \cup S$

A	B
0	1

(d) $R \cap S$

A	B
2	3

(e) $R - S$

图8-11　关系代数运算的示例

8.7.3　选择运算符

关系代数中的其他运算符是用来执行本章中研究的这些操作的。例如，我们经常想从关系中提取出满足某些条件的元组，比如从“课程–学号–成绩”关系中提取出“课程”组分为CS101的所有元组。为达成这一目的，要使用选择运算符。该运算符接受一个关系作为操作数，但还要接受一个条件表达式作为“参数”。我们把选择运算符写为$\sigma_C(R)$，其中σ（小写的希腊字母西格玛）是表示选择的符号，C是条件，而R是关系操作数。条件C可以用关系R的模式中的属性以及常数作为操作数。条件C可以使用的运算符就是常用于C语言条件表达式中的那些，也就是算术比较符和逻辑连接符。

这一运算的结果是模式与R的模式相同的关系。我们要把在将条件C中的属性A替换为元组t对应列A的组分时使得条件C为真的每个元组t都放入该关系中。

✦　示例 8.12

设CSG表示图8-1中的“课程–学号–成绩”关系。如果想要那些课程组分为“CS101”的元组，就可以写出如下表达式

$$\sigma_{\text{课程}=\text{“CS101”}}(CSG)$$

这一表达式的结果是模式与CSG相同，也就是具有{课程，学号，成绩}模式的关系，而且元组的集合就如图8-12所示。也就是说，只有这些“课程”组分为CS101的元组才能使条件为真。这样一来，当我们用CS101替换了“课程”，条件就成了CS101 = CS101。如果该元组的“课程”组分有其他的值，比如EE200，就得到EE200 = CS101这样的不成立的表达式。

课　程	学　号	成　绩
CS101	12345	A
CS101	67890	B
CS101	33333	A–

图8-12　表达式$\sigma_{课程=\text{“CS101”}}(CSG)$的结果

8.7.4 投影运算符

选择运算符会生成某关系删除若干行之后的副本，而我们经常想要生成关系删除若干列之后的副本。为达到这一目的，我们还有用符号π表示的投影运算符。和选择一样，投影运算符也是接受一个关系作为参数，它还要接受另一个参数，就是从作为参数的关系的模式中选取的属性列表。

如果R是具有属性集合$\{A_1, \cdots, A_k\}$的关系，而$(B_1, \cdots, B_n)$是某些A组成的列表，那么$\pi_{B_1, \ldots, B_n}(R)$，关系$R$到属性$B_1, \cdots, B_n$上的投影，是按照以下方式形成的元组集合。取$R$中的元组$t$，提取其属性$B_1, \cdots, B_n$中的组分，假设这些组分分别是$b_1, \cdots, b_n$，然后将元组$(b_1, \cdots, b_n)$添加到关系$\pi_{B_1, \ldots, B_n}(R)$中。请注意，$R$中可能有不止一个元组在$B_1, \cdots, B_n$中的组分都相同。如果这样的话，这些元组的投影只有一个副本会进入$\pi_{B_1, \ldots, B_n}(R)$，因为该关系和所有关系一样，不可能含有某一元组的多个副本。

✦ 示例 8.13

假设只想看到选修CS101课程的学生的学号。可以应用与示例8.12相同的选择运算，这样就给出了CSG关系中所有对应CS101的元组，不过之后还必须将课程和成绩投影掉，也就是只投影到“学号”上。执行这两项运算的表达式为

$$\pi_{学号}\left(\sigma_{课程=\text{"}CS101\text{"}}(CSG)\right)$$

该表达式的结果是图8-12的关系投影到其“学号”组分上，也就是如图8-13所示的一元关系。

学　号
12345
67890
33333

图8-13　选修CS101的学生

8.7.5 关系的联接

最后，我们需要一种方式，用来表示两个关系被关联起来从而可以从一个关系向另一个关系导航的概念。为了达成这一目的，要使用表示为$\bowtie$的联接运算符。[①]假设有两个关系R和S，其属性集合（模式）分别为$\{A_1, \cdots, A_n\}$和$\{B_1, \cdots, B_m\}$。我们从两个集合中各选出一个属性，比方说是A_i和B_j，而这些属性将成为以R和S为参数的联接运算的参数。

① 我们这里描述的“联接”不如关系代数中常见的联接运算更一般化，但它可以用来让我们从这一运算符受益，而不用深入到该主题的所有复杂性中。

要形成R和S的写作 $R \underset{A_i=B_j}{\bowtie} S$ 的这种联接，就要从R中取出各元组r，并从S中取出各元组s加以比较。如果元组r对应A_i的组分等于元组s对应B_j的组分，就从r和s形成一个元组，否则，就不会从r和s的配对中创建元组。要从r和s形成元组，就要取r中的组分，后面加上s中的所有组分，但要去掉对应B_j的组分，因为它和r中对应A_i的组分是相同的。

关系 $R \underset{A_i=B_j}{\bowtie} S$ 就是上述方式所形成的元组构成的集合。请注意，若R的A_i列和S的B_j列都没有值出现，则这一关系也可能不含任何元组。在另一种极端情况下，R中每个元组A_i组分的值都相同，而该值也出现在S中每个元组的B_j组分里。那么，联接得到的关系中元组的数量就等于R中元组数量乘以S中元组数量的积，因为元组的每个有序对都能匹配。一般而言，真相就藏在这些极端情况之间的某个地方，R中的各元组与S中的一些（而非全部）元组配对。

联接得到的关系的模式是$\{A_1, \cdots, A_n, B_1, \cdots, B_{j-1}, B_{j+1}, \cdots, B_m\}$，也就是R和S中除$B_j$之外的全部属性构成的集合。不过，还是可能有属性重名的情况出现，如果说是A中的某一属性与B中（除B_j之外，不是要联接的属性）的某一属性相同。如果出现这种情况，这一对相同的属性中就必须有一个要重命名。

✦ 示例 8.14

假设我们要对“课程–日子–时刻”关系（简称CDH）和“课程–教室”关系（简称CR）进行连接。例如，我们可能想知道每间教室都有哪些时间是在上课的。要回应这一查询，就必须将来自CR的各元组与来CDH的各元组配对，要求配对的两个元组的课程组分是相同的，也就是说，这两个元组说的是相同的课程。因此，如果在要求二者的“课程”属性相等的情况下联接CR和CDH，就会得到具有{课程，教室，日子，时刻}模式的关系，它所含的元组(c, r, d, h)满足(c, r)是CR的元组且(c, d, h)是CDH的元组这两个条件。定义该关系的表达式为

$$CR \underset{\text{课程}=\text{课程}}{\bowtie} CDH$$

假设CR和CDH关系含有图8-2中的那些元组，那么该表达式产生的关系的值就如图8-14所示。

课　程	教　室	日　子	时　刻
CS101	Turing Aud.	M	9AM
CS101	Turing Aud.	W	9AM
CS101	Turing Aud.	F	9AM
EE200	25 Ohm Hall	Tu	10AM
EE200	25 Ohm Hall	W	1PM
EE200	25 Ohm Hall	Th	10AM

图8-14　课程 = 课程上CR和CDH的联接

要知道图8-14中的关系是如何构建的，就要考虑一下CR的第一个元组，(CS101，Turing Aud.)。我们要检查CDH中那些“课程”值也是CS101的元组。在图8-2c中，可以看到前3个元组都是能匹配的，而且我们可以由这些元组构建图8-14的前3个元组。例如，CDH的第一个元组（CS101，M，9AM）与元组（CS101，Turing Aud.）联接，就得到了图8-14的第一个元组。要注意该元组是如何与构成它的两个元组各自对应的。

同样，CR的第二个元组（EE200，25 Ohm Hall）与CDH的后3个元组有着相同的“课程”组分。这3个配对就构成了图8-14中的后3个元组。而CR的最后一个元组（PH100，Newton Lab.）的“课程”组分与CDH任一元组的“课程”组分都不同。因此，该元组没有对这一联接作出任何贡献。

8.7.6　自然联接

当我们联接两个关系R和S时，常会遇到要等同的属性同名的情况。如果此外还有R和S没有其他属性同名，那么就可以将联接的参数省略，将其简写为$R \bowtie S$。这样的联接就叫作*自然联接*。

例如，示例8.14中的联接就是自然联接。要等同的属性都叫“课程”，而CR和CDH中其他属性的名称都是不同的。因此我们可以将该联接简写为$CR \bowtie CDH$。

8.7.7　关系代数表达式的表达式树

就像为算术表达式绘制表达式树那样，可以将关系代数表达式表示为树。叶子都是用操作数标记，也就是用特定的关系或是表示关系的变量来标记。每个内部节点都是由运算符标记的，如果是选择、投影或联接（除了不需要参数的自然联接）运算符，还要包括运算符的参数。各内部节点N的子节点都是表示应用了节点N处的运算符的操作数。

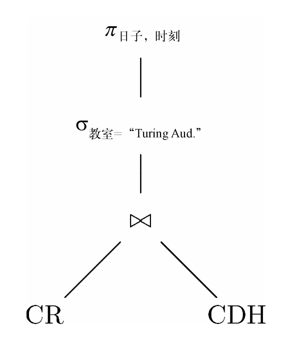


图8-15　关系代数中的表达式树

✦ 示例 8.15

接着示例8.14的情况，假设我们想知道的不是整个$CR \bowtie CDH$关系，而只是想知道在Turing Aud.有课的“日子–时刻”对。然后我们需要取图8-14中的关系，并进行下列操作。

(1) 选择那些“教室”组分为“Turing Aud.”的元组；

(2) 将这些元组映射到日子和时刻属性上。

按上述次序执行这一系列联接、选择和投影的表达式为

$$\pi_{\text{日子，时刻}}\Big(\sigma_{\text{教室=“Turing Aud”}}(CR \bowtie CDH)\Big)$$

我们可以将这一表达式表示成如图8-15所示的树。在表示联接的节点处计算出的关系就是图8-14所示的关系。而选择节点处的关系是图8-14中的前3个元组，因为它们的“教室”组分中都有Turing Aud.。该表达式树根节点对应的关系如图8-16所示，也就是这3个元组的日子和时刻组分。

日　子	时　刻
M	9AM
W	9AM
F	9AM

图8-16　图8-15中表达式的结果

SQL，基于关系代数的语言

很多现代数据库系统都使用一种名为SQL（Structured Query Language，结构化查询语言）的语言表示查询。尽管对该语言的详细介绍超出了本书范围，不过我们在这里可以用几个例子来给读者一点关于SQL的印象。

```
SELECT StudentId
FROM CSG
WHERE Course = "CS101"
```

就是用SQL的方式表示示例8.13的查询，也就是

$$\pi_{\text{学号}}\left(\sigma_{\text{教室}=\text{“CS101”}}(CSG)\right)$$

其中FROM子句表示该查询的对象关系，而WHERE子句给出了选择的条件，SELECT子句则给出了答案投影到的那些属性。很不幸的是，SQL中的关键词SELECT并非对应关系代数中的选择运算符，而是对应投影运算符。

举个更复杂的例子，可以将示例8.15中查询$\pi_{\text{日子, 时刻}}\left(\sigma_{\text{教室}=\text{“Turing Aud”}}(CR \bowtie CDH)\right)$表示为如下SQL程序。

```
SELECT Day, Hour
FROM CR, CDH
WHERE CR.Course = CDH.Course AND Room = "Turing Aud."
```

这里的FROM子句告诉我们要联接*CR*和*CDH*这两个关系。WHERE子句的第一部分是联接的条件，它表示*CR*的课程属性必须和*CDH*的课程属性相等；WHERE子句的第二部分则是选择的条件；而SELECT子句则告诉我们映射中的那些属性。

8.7.8 习题

(1) 用关系代数表示8.4节习题(2)中(a)(b)(c)小题的查询，假设我们想要的答案是完整的元组。

(2) 重复习题(1)的练习，假设想要的只有那些规范中带*的组分。

(3) 用关系代数表示8.6节习题(2)中(a)(b)(c)小题的查询。请注意(c)小题，在将关系与其自身联接时必须要重命名一些属性。

(4) 用关系代数表示“C.Brown星期一上午9点在哪里”这一查询。8.6节最后的讨论应该能指示出回应该查询所必需的联接。

(5) 画出习题(2)中(a)至(c)的情况、习题(3)中(a)至(c)的情况以及习题(4)中的查询所对应的表达式树。

8.8 关系代数运算的实现

为关系代数运算使用得当的数据结构和算法可以加快数据库查询的速度。在本节中，我们将考虑一些相对简单常见的关系代数运算的实现策略。

8.8.1 并交差的实现

这3种基本集合运算的实现方式与在关系和集合中的实现方式相同。可以按照7.4节讨论过的，通过为两个集合排序与合并取两个集合或关系的并集。而交集和差集则可利用相似的技巧求得。如果参加运算的两个关系各含n个元组，就要花$O(n\log n)$的时间为其排序并用$O(n)$的时间合并，或者说总共需要$O(n\log n)$的时间。

不过，为关系R和S求并集的方式还有很多，而且有些方式还更高效。首先，我们可能不去考虑为同时出现在R和S中的元组消除重复副本的事。就是生成R的副本，比如说，放进链表中，然后将S的所有元组也添加进该链表，而不去检查S中的元组是否也出现在R中。这一操作可以在与R和S的大小之和成比例的时间内完成。而这样做的缺点在于，严格地讲，得到的结果并不是R和S的并集，因为其中可能存在重复的元组。不过，也许这些重复的存在并无大碍，因为可预期它们很少出现。或者，我们可能发现在后续的阶段中消除这些重复会更方便，比如在取更多关系的并集后进行排序，然后再消除重复。

另一种选择是使用索引。例如，假设R具有属性A上的索引，而该属性是S的键。那么如果要取二者的并集R∪S，首先从S的元组开始，并依次检查R的每个元组t。我们会在组分A中找到t的值——比方说是a，并使用该索引查找S中A组分的值也为a的元组。如果S中的这一元组与t相同，就不要再将t第二次放入并集中，而如果S中不存在键的值为a的元组，或者键值为a的元组与t不同，就要将t加入并集中。

如果索引提供了在给定元组键值的情况下每个元组平均为$O(1)$的元组查询时间，那么这种方法求并集的平均时间就与R和S的大小之和成比例。此外，只要R和S都没有重复，那么得到的关系也是没有重复的。

8.8.2 投影的实现

原则上讲，在执行投影运算时，只能检验完每个元组，并略去那些与未出现在投影列表中的属性对应的组分。索引是一点忙都帮不上的。此外，在计算了各元组的投影后，我们可能发现会留下很多重复。

例如，假设有模式为{A，B，C}的关系R，而且要计算$\pi_{A,B}(R)$。尽管R中的元组不可能A、B、C属性全都相同，但还是可能有很多元组的属性A和B会相同而只是对应属性C的值不同。这样一来，这些元组在投影中全都会得到相同的元组。

因此，在为某关系R和一列属性L计算了$S=\pi_L(R)$这样的投影后，必须消除重复。例如，可以为S排序，然后以排序的次序检查所有元组。那些在次序上与前一个元组相同的元组都要被删除。另一种消除重复的方式是将关系S看作普通集合。每当我们通过把R的元组投影到表L中的属性上生成一个元组，就将其插入该集合中。就像所有向集合插入元素的操作那样，如果待插入的元素已经在集合中，就不用做任何事情。散列表这样的结构就很适合表示由投影生成的元组构成的集合S。

如果关系R中有n个元组，那么要在消除重复前为关系S排序所需的时间为$O(n\log n)$。而如果改为在生成S的元组时对其执行散列操作，而且使用数量与n成比例的散列表元，那么整个投影运算平均要花$O(n)$的时间。因此，散列通常要略优于排序。

8.8.3 选择的实现

在执行选择运算$S=\sigma_C(R)$而且没有R上的索引时，就只能检查R中的所有元组以应用条件C。不管如何执行选择，只要R中没有重复，得到的S中就不会有重复。

不过，如果R上存在若干索引，就可以利用其中某一索引直接找到满足条件C的元组，因此就可以避免查看大多数或是所有不满足条件C的元组。条件C形如$A=b$时的情况最简单，其中A是R的某一属性，而b是某常量。如果R具有A上的索引，就可以通过在索引中查找b来检索满足该条件的所有元组。

如果条件C是若干条件的逻辑AND，那么可以利用其中某一条件查找使用了索引的元组，然后检查检索到的这些元组，看看有哪些满足其余的条件。例如，假设条件C是

$$(A = a)\ \text{AND}\ (B = b)$$

如果这A上的索引或是B上的索引中有某一个或全都存在，就可以选择使用某一个索引。假设有B上的索引，而且要么A上没有索引，要么是我们主动选择使用B上的索引。这样一来，我们就会得到关系R中B组分的值为b的所有元组。这些元组中A组分为a的都属于选择运算的结果——关系S，而检索到的其他元组则不属于S。这一选择运算所花的时间是与B组分的值为b的元组数量（通常在R中的元组数和S中的元组数之间）成比例的。

8.8.4　联接的实现

假设我们想要对模式为$\{A,B\}$的关系R和模式为$\{B,C\}$的关系S进行自然联接。还假设该联接是两个关系的B属性之间存在相等关系的自然联接。[①]如何执行这一联接取决于我们能找到属性B上的何种索引。该问题类似于我们在8.6节中讨论过的那些，当时我们是考虑如何在关系间导航，而导航的本质就是联接。

有一种直观而缓慢的联接计算方式，叫作*嵌套循环联接*。我们会按照如下方式对一个关系中的每个元组与另一关系中的每个元组加以比较。

```
for R中的各元组r do
    for S中的各元组s do
        if r 和 s 的 B 属性相同 then
            打印结合了r和s的
            属性A、B、C的元组;
```

然而，还有很多更高效的联接方式。*索引联接*就是其中之一。假设S有属性B上的索引。那么可以访问R的各元组t，并找到其B组分，比方说是b。在S的索引中查找b，这样就能得到B的值能与t匹配的所有元组。

同样，如果R有属性B上索引，就可以浏览S的所有元组。对S中的各元组，我们会使用R的B索引查找与之对应的R的元组。如果R和S都有属性B上的索引，就要任选其一来用。正如我们即将看到的，这会给联接运算所花的时间带来变化。

如果没有属性B上的索引，利用*排序联接*还是能比嵌套循环联接做得更好。首先要将R和S中的元组合并在一起，不过要重新组织这些元组，使得B组分成为所有元组的第一个组分，而且要为它们加上一个额外的组分，如果该元组来自关系R，就加上R，而如果该元组来自关系S，就加上S。也就是说，来自关系R的元组(a,b)就成了(b,a,R)，而来自关系S的元组(b,c)则成了(b,c,S)。

我们根据第一个组分（也就是b）来为合并后的元组表排序。虽然因为B值相同而联接的两个关系中元组可能混合在一起了，但是这些元组现在已经是按着次序连续排列了。[②]我们会沿着已排序表向下，依次访问具有各给定B值的元组。当到达B值为b的元组时，就可以将R中所有这

① 这里展示的两个关系分别只有一个属性（分别为A和C）没有涉及联接，不过这里提到的想法显然可以推广到具有很多属性的关系。

② 我们可以在排序的同时考虑对最后一个元组（也就是关系名）加以安排，使得来自关系R的具有给定B值的元组一定会位于来自S有着相同B值的元组之前。这样一来，对那些B值相同的元组而言，来自R的会先出现，然后是来自S的那些。

样的元组与S中这样的元组配对。因为这些元组都有着相同的B值，所以它们都要联接，而且生成联接后关系中元组所花的时间是与生成的元组数成比例的，除非是出现R中没有元组或S中没有元组的情况。即便是在R中没有元组或S中没有元组的情况下，所花时间仍然与B值为b的元组的数量成比例，为的是检查这些元组并在已排序表中跳过这些元组。

✦ 示例 8.16

假设要联接图8-2c所示的*CDH*关系与图8-2d所示的*CR*关系。在这里，课程属性就扮演着属性B的角色，日子和时刻属性则一起扮演属性A的角色，而教室属性就是属性C。*CDH*的6个元组和*CR*的3个元组首先会各自加上其关系名称。这里不需要重新排列组分，因为两个关系中课程属性都是排在第一位的。当我们对元组加以比较时，首先要比较课程组分，利用词典次序确定哪个课程名称的次序靠前。如果不分先后，也就是说，如果课程名称相同，就要比较最后一个组分，其中*CDH*要先于*CR*。如果还是不分先后，就可以让其中任意一个元组先于另一个元组。

已排好序的元组如图8-17所示。请注意，该表并非关系，因为它所含元组的长度不尽相同。不过，它将对应CS101的元组和对应EE200的元组组织在一起，这样一来就可以很容易地联接这些元组分组了。

CS101	M	9AM	*CDH*
CS101	W	9AM	*CDH*
CS101	F	9AM	*CDH*
CS101	Turing Aud.	*CR*	
EE200	Tu	10AM	*CDH*
EE200	W	1PM	*CDH*
EE200	F	10AM	*CDH*
EE200	25 Ohm Hall	*CR*	
PH100	Newton Lab.	*CR*	

图8-17 *CDH*和*CR*中所有元组构成的已排序表

8.8.5 联接方法的比较

假设要联接模式为$\{A, B\}$的关系R和模式为$\{B, C\}$的关系S，并设R和S分别有r个元组和s个元组。还有，设联接中的元组数为m。要记住，如果R的每个元组都与S中的每个元组（因为它们都有相同的B值）联接，那么m可以有rs那么大，而如果R中没有元组的B值与S中元组的B值相等，那么m还可以小到0这么小。最后，假设可以在平均$O(1)$的时间内查找任意索引中的任意值，就像索引是有着相当大量的散列表元的散列表时能做到的那样。

每种联接方法生产输出都至少要花$O(m)$的时间。不过，有些方法所花的时间要更多一些。如果使用嵌套循环联接，就要花rs的时间执行比较。因为$m \leqslant rs$，所以可以忽略生成输出所花的时间，并说配对所有元组的时间开销为$O(rs)$。

另一方面，我们可以为这些关系排序。如果使用类似归并排序的算法为含$r+s$个元组的表排序，所需的时间就是

$$O\big((r+s)\log(r+s)\big)$$

要从已排序表中毗连的元组构建输出元组，就要花$O(r+s)$的时间检查该表，还要花$O(m)$

的时间生成输出。排序所需的时间主导了$O(r+s)$项，不过生成输出所花的$O(m)$既可能大于也可能小于排序的时间。因此通过排序进行联接的算法的运行时间必须包含这两项，这一运行时间就是

$$O\big(m+(r+s)\log(r+s)\big)$$

因为m从不会大于rs，而且$(r+s)\log(r+s)$只有在一些少见的情况下才大于rs（比如，r或s为0时），所以可以说排序联接一般而言要比嵌套循环联接更快。

现在假设有关系S中属性B上的索引。要花$O(r)$的时间查看R的各元组并在索引中查找这些元组的B组分的值。这时我们必须加上检索对应各B值的匹配元组以及生成输出元组的时间开销$O(m)$。因为m既可以大于r也可以小于r，所以表示该索引联接时间开销的表达式就是$O(m+r)$。同样，如果有关系R中属性B上的索引，就可以在$O(m+s)$的时间内执行该索引联接。因为除了某些罕见的情形（比如$r+s\leqslant 1$）之外，r和s都小于$(r+s)\log(r+s)$，所以索引联接的运行时间要小于排序联接的时间。当然，如果想要进行索引联接，就需要联接中所涉及的某一属性上的索引，而排序联接则可以对任意关系进行。

8.8.6　习题

(1) 假设图8-2a所示的“学号−姓名−地址−电话”关系（*SNAP*）具有学号属性（键）上的主索引，而且有电话属性上的辅助索引。如果条件C分别为以下3种，那么我们该如何最高效地为查询$\sigma_C(SNAP)$计算回应？

(a) 学号 = 12345　AND 地址 ≠ “45 Kumquat Blvd”

(b) 姓名 = “C.Brown”AND 电话 = 555-1357

(c) 姓名 = “C.Brown”OR 电话 = 555-1357

(2) 说明如何通过为示例8.16中合并的元组表排序，来为图8-1中的*CSG*关系与图8-2a中的*SNAP*关系进行排序联接。假设是想进行自然联接，或者说是想要“学号”组分上的相等关系。给出排序的结果，就像图8-17那样，并给出联接运算得到的关系中的元组。

(3) * 假设要联接关系R和S，它们各含n各元组，而且结果中有$O(n^{3/2})$个元组。分别写出利用以下各项技术进行联接时大O运行时间的公式，将其表示为n的函数。

(a) 嵌套循环联接。

(b) 排序联接。

(c) 索引联接，使用R的联接属性上的索引。

(d) 索引联接，使用S的联接属性上的索引。

(4) * 我们提出过利用作为某关系的键的属性A上的索引为两个关系取并集。如果具有索引的属性A不是键，这是否仍为合理的取并集方式？

(5) * 假设想使用R和S二者之一的某属性A上的索引计算(a) $R\cap S$；(b) $R-S$。能否取得与两个关系大小之和接近的运行时间？

(6) 如果要将关系R投影到含有R的键的属性集合上，是否需要消除重复？为什么？

8.9　关系的代数法则

就像其他代数那样，通过对表达式进行变形，通常能“优化”表达式。也就是说，我们可以接受一个求值开销很大的表达式，并将其转换成求值开销较小的等价表达式。对算术或逻辑表达式的变形有时能节省一些运算，而对关系代数表达式进行合适的变形，可以节省几个数量

级的求值时间。因为优化过的和未优化的关系代数表达式在运行时间上有着巨大的差异，所以如果程序员要用非常高级的语言（比如我们在8.7节中提过的SQL语言）编程，优化这种表达式的能力就是很关键的。

8.9.1 涉及并交差的法则

7.3节涵盖了用于集合并交差运算的主要代数法则。这些法则也能应用到集合的特例，关系上，不过读者应该记住关系模型的要求，就是运算所涉及关系的模式必须相同。

8.9.2 涉及联接的法则

从某种意义上讲，联接运算符是可交换的，而从另一种意义上讲，它又不是可交换的。假设要计算自然联接$R \bowtie S$，其中R具有属性A和B，而S具有属性B和C。那么$R \bowtie S$的模式中的各列按次序排列就是A、B、C。如果是求$S \bowtie R$，可以得到本质上相同的元组，不过各列的次序是B、C、A。因此，如果我们坚信各列的次序并非无关紧要，那么联接就不具交换性。不过，如果我们认可连带列名一起整列交换的关系其实是相同的关系，就可以认为联接是可交换的，在这里我们要采纳这一观点。

联接运算符并不总是符合结合律。例如，假设有模式分别为$\{A, B\}$、$\{B, C\}$和$\{A, D\}$的3个关系R、S和T。假设要取自然联接$(R \bowtie S) \bowtie T$，其中首先要让R和S的B组分相等，接着让结果的A组分与关系T的A组分相等。如果是从右边关联，就得到$R \bowtie (S \bowtie T)$。关系S和T的模式分别为$\{B, C\}$和$\{A, D\}$。没有办法选择令其相等以达到自然联接效果的属性对。

不过，在某些条件下结合律对联接运算来说是成立的。我们将如下公式的证明留给读者作为练习

$$\left((R \underset{A=B}{\bowtie} S) \underset{C=D}{\bowtie} T\right) \equiv \left(R \underset{A=B}{\bowtie} (S \underset{C=D}{\bowtie} T)\right)$$

只要A是R的属性，B和C是S的两个不同属性，而D是T的属性就行。

8.9.3 涉及选择的法则

关系代数最实用的法则涉及选择运算符。如果选择的条件就像实践中常见的那样是要求指定的组分有特定的值，则选择运算的结果关系中的元组数要比原关系的元组数少很多。因为一般而言当运算应用到较小的关系上时所花的时间较少，所以尽可能早地应用选择运算是极为有利的。就代数学而言，如果想早点应用选择运算，就要动用代数法则让选择运算符沿着表达式树向下传递，达到其他运算符下方。

这种法则的一个例子就是

$$\left(\sigma_C(R \bowtie S)\right) \equiv \left(\sigma_C(R) \bowtie S\right)$$

只要条件C中提到的属性都是关系R的属性，它就成立。同样，如果条件C提及的所有属性都是S的属性，就可以使用如下法则将选择运算符下压到S

$$\left(\sigma_C(R \bowtie S)\right) \equiv \left(R \bowtie \sigma_C(S)\right)$$

这两条法则都称为*选择的下压*（selection pushing）。

当选择中的条件很复杂时，可能要用一种方式下压一部分，而用另一种方式下压另一部分。为了将选择分割为两个或多个部分，就需要法则

$$\sigma_{C \text{ AND } D}(R) \equiv \sigma_C\left(\sigma_D(R)\right)$$

请注意，如果这些分割出的部分用AND相连，就只能将条件分为两个部分——这里是C和D。直觉上讲，当我们为C和D这两个条件的AND进行选择时，要么是检查关系R中的所有元组，看看这些元组是否同时满足C和D，要么是检查R的所有选择，选出那些满足条件D的，然后再检查满足条件D的元组，看看其中有哪些满足条件C。我们将该法则称为**选择的分割**（selection splitting）。

另一种必要的法则是选择的交换律。如果要对某关系应用两次选择，那么应用这些选择的次序是无关紧要的，选出的元组都会是相同的。我们可以正式地将其写为，对任何条件C和D，有

$$\sigma_C\left(\sigma_D(R)\right) \equiv \sigma_D\left(\sigma_C(R)\right)$$

✦ 示例 8.17

我们再来看看8.6节中考虑过的复杂查询："C.Brown星期一上午9点在哪里？"这一查询涉及4个关系上的导航：

(1) *CSG*（课程–学号–成绩）；

(2) *SNAP*（学号–姓名–地址–电话）；

(3) *CDH*（课程–日子–时刻）；

(4) *CR*（课程–教室）。

为了得出表示该查询的代数表达式，首先可以求这4个关系的自然联接。也就是通过让学号组分相等连接*CSG*和*SNAP*。可以把该运算视为对"课程–学号–成绩"元组进行扩展，为该元组加上所提及学生的姓名、地址和电话号码。当然，我们不会希望用这种方式存储数据，因为这会迫使我们为某学生选修的每门课程将该学生的这些信息重复一遍。不过，我们并不是要存储这些数据，而只是要设计表达式计算它。

通过让课程组分相等，我们要将$CSG \bowtie SNAP$的结果与*CDH*联接。这次联接会取各*CSG*元组（已经扩展了学生信息），为每个上课时段制作一个副本，并将各元组扩展为具有一对可能"日子–时刻"值。最后要将$(CSG \bowtie SNAP) \bowtie CDH$的结果与*CR*联接，还是让课程组分相等，结果就是通过添加带有某一上课时段所在教室的组分扩展了各元组。得到的关系模式为：

{课程，学号，成绩，姓名，地址，电话，日子，时刻，教室}

而元组(c，s，g，n，a，p，d，h，r)的含义就是：

(1) 学生s选修了课程c并拿到了g的成绩；

(2) 学号为s的学生的姓名是n，他（她）的地址是a而且电话号码为p；

(3) 课程c是在教室r上课，而且该课程某一次上课时间是d日的h时。

对该元组集合，必须应用将考量限制到相关元组的选择，也就是姓名组分为"C.Brown"，日子组分为"M"，而且"时刻"组分为"9AM"的元组。假设C.Brown最多选修了一门在星期一上午9点上课的课程，这样的元组就最多只有一个。因为我们想要的回应是该元组的"教室"组分，所以要通过投影到"教室"属性上来完成表达式。表示这一查询的表达式树如图8-18所示。它由4路联接组成，接着是选择，然后是投影。

如果按照图8-18这样的写法为该表达式求值，就要通过联接*CSG*、*SNAP*、*CDH*和*CR*构建一个巨大的关系，然后将其限制到一个元组，再将该元组投影到一个组分上。请记住，由8.6节我们可知并不一定要构建如此庞大的关系，而是可以"将选择沿着树向下压"，从而限制联接运算

中涉及的关系，因此就大大限制了我们要构建的关系的大小。

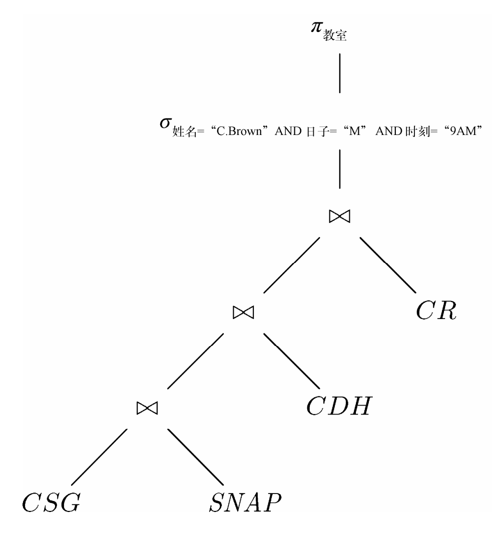


图8-18 确定C.Brown星期一上午在哪里的初始表达式

第一步如图8-19a所示。请注意，选择只涉及姓名、日子和时刻属性。它们中没有一个来自图8-18顶层联接的右操作数，而都来自左侧，也就是*CSG*、*SNAP*和*CDH*的联接。因此可以将选择压到顶层联接之下，并只将其应用到该运算的左操作数，一如我们在图8-19a中所见。

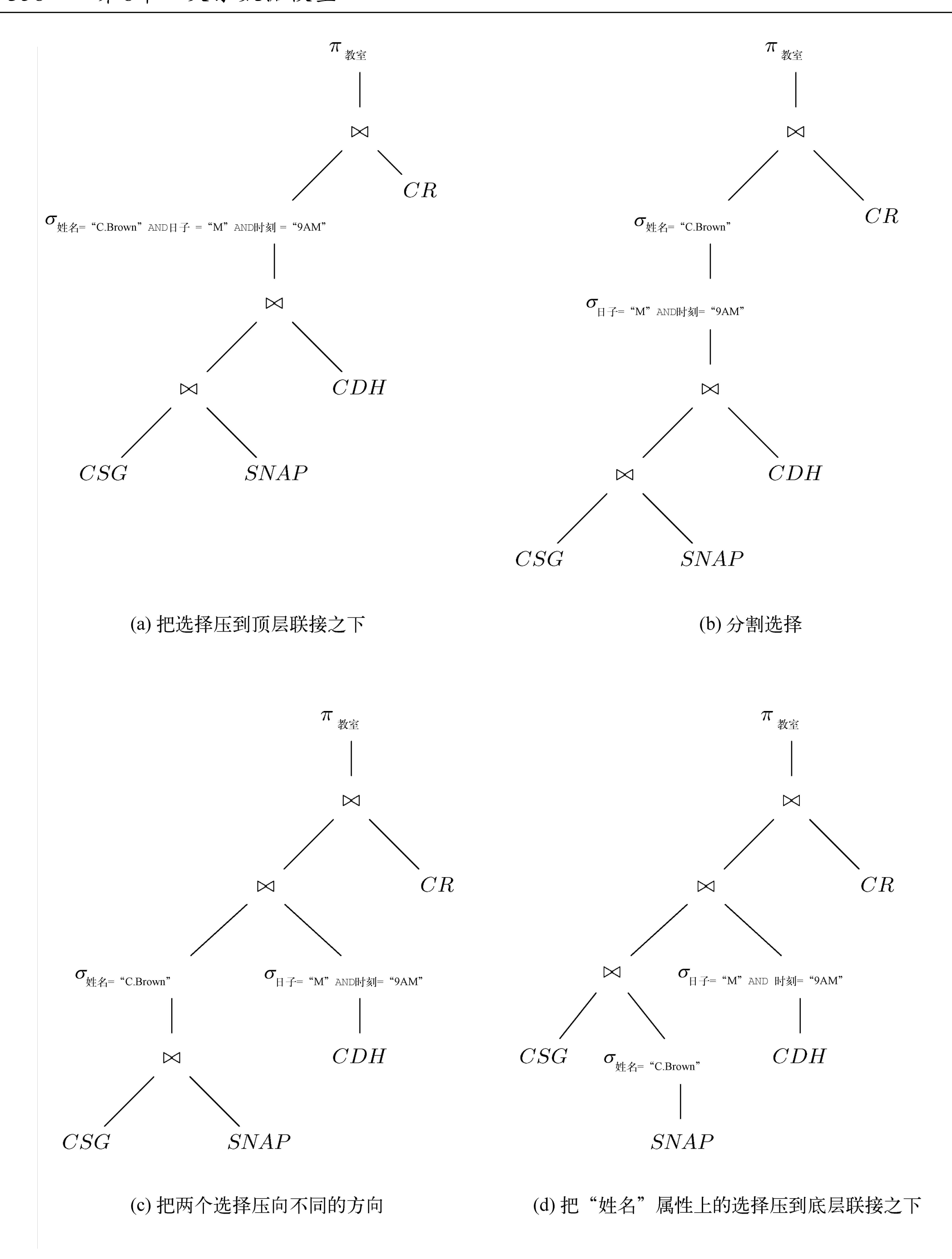
现在就没法继续下压选择运算符了，因为所涉及的姓名属性来自图8-19a中层联接的左操作数，而其他两个属性，日子和时刻，则来自其右操作数，*CDH*关系。因此必须分割选择中的条件，它是3项条件的AND，可以分割成3个选择，不过在本例中只要把姓名 = C.Brown这一条件与其他两个分开即可，分割的结果如图8-19b所示。

现在，涉及日子和时刻的选择可以下压到中层联接的右操作数，因为右操作数是同时具有日子和时刻属性的*CDH*关系。然后另一个涉及姓名的选择就可以下压到中层联接的左操作数，因为该操作数是$CSG \bowtie SNAP$，含有姓名属性。这两项改变会带来如图8-19c所示的表达式树。

最后，姓名上的选择涉及*SNAP*的属性，因此可以将该选择下压到底层联接的右操作数。这一改变如图8-19d所示。

现在该表达式给我们的计划与8.6节中为该查询设计的计划几乎是相同的。首先从图8-19d总的表达式底层开始，找到名为C.Brown的学生的学号。把姓名 = C.Brown的*SNAP*元组与*CSG*关系联接，得到C.Brown选修的课程。当我们把第二次选择应用到*CDH*关系时，就得到星期一上午9点上课的课程。因此图8-19d所示的中层联接给了我们C.Brown选修了而且在星期一上午9点上课的课程。而顶层联接则得到这一时间这门课程上课的教室，而这里的投影就给出了这些教室作为回应。

这一计划与8.6节中的计划主要的差别在于，后者是先将元组中无用的组分投射走，而这里的计划则是要带着这些组分直到最后。因此要完成对关系代数表达式的优化，就需要可以将投影沿着树向下压的法则。正如我们将在8.9.4节中看到的，这些法则与用于选择的法则不尽相同。



(a) 把选择压到顶层联接之下　　(b) 分割选择

(c) 把两个选择压向不同的方向　　(d) 把“姓名”属性上的选择压到底层联接之下

图8-19　将选择运算下压

8.9.4　涉及投影的法则

首先，与选择可以被压到并集、交集或差集之下（只要我们将选择同时压向两个操作数）不同的是，投影只能压到并集之下。也就是，法则

$$\big(\pi_L(R\cup S)\big)\equiv\big(\pi_L(R)\cup\pi_L(S)\big)$$

成立。不过，$\pi_L(R \cap S)$ 不一定等同于。例如，假设R和S都是模式为{A，B}的关系，R只包含元组(a，b)，而S只包含元组(a，c)。那么$\pi_A(R) \cap \pi_A(S)$ 只含（单组分）元组(a)，而$\pi_A(R \cap S)$ 则不含元组（因为 $R \cap S$ 为空）。因此就有了的情况。

$$\left(\pi_A(R \cap S)\right) \neq \left(\pi_A(R) \cap \pi_A(S)\right)$$

$$\pi_L(R) \cap \pi_L(S)$$

将投影压到联接以下是有可能的。一般而言，我们需要为联接运算的每个操作数都加上投影运算符。如果有表达式$\pi_L\left(R \underset{A=B}{\bowtie} S\right)$，那么所需的$R$的属性是那些出现在属性表$L$中的属性，以及联接运算所依据的来自关系$R$的属性$A$。同样，我们需要$S$中那些在属性表$L$上的属性，以及联接依据的属性$B$，不管$B$是否在$L$中。正式地讲，将投影压到联接之下的法则是

$$\left(\pi_L\left(R \underset{A=B}{\bowtie} S\right)\right) \equiv \left(\pi_L\left(\pi_M(R) \underset{A=B}{\bowtie} \pi_N(S)\right)\right)$$

其中

(1) 表M是由L中那些在R模式中的属性构成，如果属性A不在L中，就还要加上A；

(2) 表N是由L中那些在S模式中的属性构成，如果属性B不在L中，就还要加上B。

要注意到，应用这一投影下压法则的实用方式是从左向右应用，即便我们因此引入了两项额外的投影而且没有减少任何运算。原因在于，尽早地投影出那些可以投影的属性（也就是将投影尽可能压到表达式树靠下的位置）通常是有利的。如果联接属性A不在表L中，我们在联接运算后可能仍然要执行到表L上的投影运算。回想一下另一个联接属性，来自S的B属性，无论如何都不会出现在联接中。

有时候，表M和（或）表N分别使用R或S的所有属性组成的。如果这样，就没理由执行这一的投影了，因为它是没有效果的，除非关系的各列可能是种毫无头绪的排列。因此我们要使用如下法则。

$$\pi_L(R) \equiv R$$

表明了表L是由R的模式中的所有属性组成的。请注意这一法则是认可“整列交换不会改变关系”这一观点的。

还有一种我们不想进行投影的情况。假设子表达式$\pi_L(R)$是某更大表达式的一部分，并设R是单一关系而不是涉及运算符的表达式。还假设在表达式树中该子表达式之上的位置还有另一个投影。现在，要执行R上的投影就要求我们检查整个关系，而不管有没有索引的存在。如果我们带着R中未在表L上的属性继续向下，直到下一次有机会将这些属性投影掉，就经常能节省大量时间。

例如，在接下来的示例中我们将讨论子表达式

$$\pi_{\text{课程，学号}}(CSG)$$

它的作用是去掉成绩。因为整个（表示示例8.17中的查询的）表达式最终要将焦点集中在CSG关系的少量元组上，所以最好是迟一些再将成绩投影走，这样做就避免了检查整个CSG关系。

✦ 示例 8.18

我们将图8-19d变成下压投影运算。根节点处的投影会首先被压到顶层联接之下。投影表只有“教室”组成，而联接运算两侧的联接属性都是“课程”。因此，在左边我们只投影到“课程”上，因为“教室”不是左侧表达式中的属性。而该联接的右操作数则要投影到“课程”和“教室”属性上。因为这两个属性都是操作数CR中的，所以可以略去这一投影。得到的表达式如图8-20a所示。

现在，可以将到“课程”属性上的投影压到中层联接之下。因为“课程”还是联接运算两边的联接属性，所以我们在中层联接下引入了两个$\pi_{\text{课程}}$运算符。由于中层联接的结果只有“课

程”属性，因此我们不再需要该联接之上的投影了，新表达式就如图8-20b所示。请注意，涉及的两个关系的元组都只有一个组分“课程”的这次联接，其实是集合的交集。这是说得通的，它是C.Brown所选的课程的集合与星期一上午9点上课的课程的集合的交集。

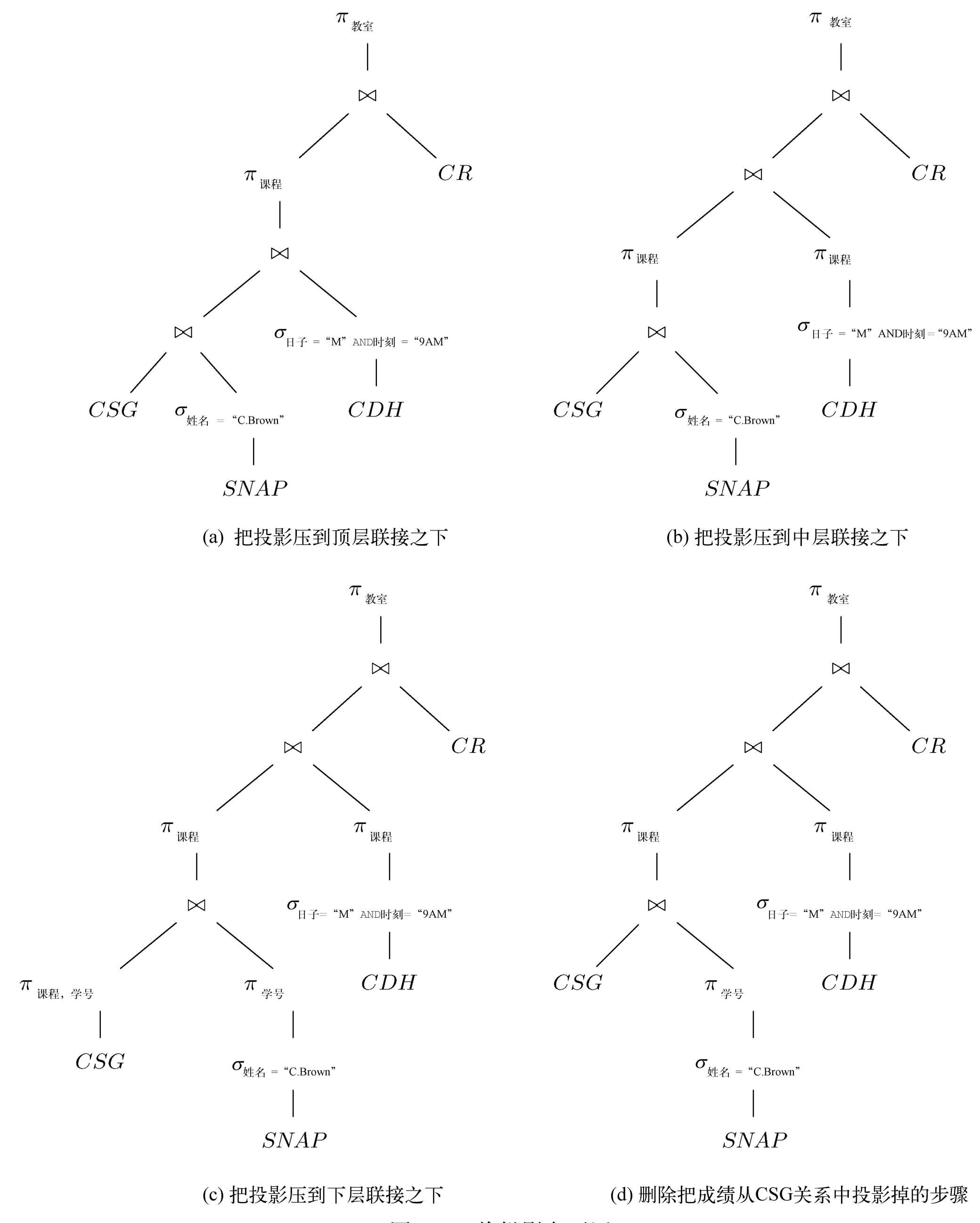


图8-20　将投影向下压

至此，我们需要将$\pi_{课程}$压到底层联接之下。两侧的联接属性都是学号，因此左侧的投影属性表就是(课程，学号)，而右侧的则就是学号，因为课程不是右侧表达式中的属性。得到的表达式如图8-20c所示。

最后，正如我们在示例之前的内容中提过的，不立即将*CSG*关系中的成绩属性投影掉是有利的。在该投影之上我们会遇到运算符$\pi_{课程}$，不管怎样它都会把成绩从中去掉。如果使用图8-20d所示的表达式，从本质上讲就有了8.6节中为这一查询设计的计划。也就是说，表达式$\pi_{学号}\left(\sigma_{姓名="C.Brown"}(SNAP)\right)$给了我们名为C.Brown的学生的学号，而后面跟着投影$\pi_{课程}$的第一次联接就给了我们那些学生选修的课程。如果关系*SNAP*关系有姓名属性上的索引，且*CSG*关系有学号属性上的索引，那么这些运算就能很快执行完。

子表达式$\pi_{课程}\left(\sigma_{日子="M" \text{ AND } 时刻="9AM"}(CDH)\right)$的值就是在星期一上午9点上课的课程，而中层联接会将这些结合求交集，以给出姓名为C.Brown的学生选修了的在星期一上午9点上课的课程。最后，后面带着投影的顶层联接会在*CR*关系中查找这些课程（如果有课程属性上的索引将会是很快的操作），并生成与之关联的教室作为回应。

8.9.5 习题

(1) * 证明：只要*A*是*R*的属性，*B*和*C*是*S*不同的属性，而且*D*是*T*的属性，就有

$$\left((R \underset{A=B}{\bowtie} S) \underset{C=D}{\bowtie} T\right) \equiv \left(R \underset{A=B}{\bowtie} (S \underset{C=D}{\bowtie} T)\right)$$

为什么 $B \neq C$ 这一条件很重要？提示：请记住，这样的属性在联接执行时会消失掉。

(2) * 证明：只要*A*是*R*的属性，*B*是*S*的属性，*C*是*T*的属性，就有

$$\left((R \underset{A=B}{\bowtie} S) \underset{A=C}{\bowtie} T\right) \equiv \left(R \underset{A=B}{\bowtie} (S \underset{B=C}{\bowtie} T)\right)$$

(3) 取8.7节习题(3)中的各关系代数查询，并将选择和投影运算下压到尽可能远的位置。

(4) 我们来对关系代数运算得到的关系中元组的数量进行如下全面简化。

(i) 每个操作数关系含有1000个元组。

(ii) 在联接分别具有*n*个和*m*个元组的关系时，得到的关系中含有 $mn/100$ 个元组。

(iii) 在执行条件为*k*个条件（每个条件都会让一种属性等于一个常数值）的AND的选择时，我们将关系的大小除以10^k。

(*iv*)在执行投影时，关系的大小不变。

此外，让我们来估计一下用为每个内部节点计算出的关系大小之和给表达式求值的开销。给出图8-18、图8-19a到图8-10d和图8-20a到图8-20d中各表达式的开销。

(5) * 证明选择的下压法则

$$\left(\sigma_C(R \bowtie S)\right) \equiv \left(\sigma_C(R) \bowtie S\right)$$

提示：要证明两个集合相等，通常最简单的方式就是如7.3节中描述的那样，证明两个集合互为对方的子集。

(6) * 证明以下法则。

(a) $\left(\sigma_C(R \cap S)\right) \equiv \left(\sigma_C(R) \cap \sigma_C(S)\right)$

(b) $\left(\sigma_C(R \cup S)\right) \equiv \left(\sigma_C(R) \cup \sigma_C(S)\right)$

(c) $\left(\sigma_C(R - S)\right) \equiv \left(\sigma_C(R) - \sigma_C(S)\right)$

(7) * 给出反例，证明下式不成立。

$$\left(\pi_L(R - S)\right) \equiv \left(\pi_L(R) - \pi_L(S)\right)$$

(8) ** 有些时候，可以利用“等价关系”

$$\sigma_C(R \bowtie S) \equiv \left(\sigma_C(R) \bowtie \sigma_C(S)\right) \quad (8.1)$$

将选择同时沿着联接的两个方向下压。

(a) 在什么情形下(8.1)式真正是等价的？

(b) 如果(8.1)式是有效的，不是将选择只压向*R*或只压向*S*，那么何时使用该法则会更合适？

8.10　小结

大家在学习过本章后应该记住以下要点。

- 被称为“关系”的二维表是一种多功能的信息存储方式。
- 关系中的行称为“元组”，而列称为“属性”。
- “主索引”将关系的元组表示为数据结构，而且会分配这些元组，使得利用某些属性(表示索引的“定义域”)的值的运算会因为这种分配变得容易。
- 关系的“键”是能唯一确定该关系其他属性的值的属性构成的集合。通常主索引会使用键作为其定义域。
- “辅助索引”这种数据结构可以简化指定了某一特定属性（通常不是主索引对应的定义域中的属性）的运算。
- 关系代数是一种高级表示法，用来表示与一个或多个关系有关的查询。其基本运算包括并、交、差、选择、投影和联接。
- 有很多实现联接的方式要比直观的“嵌套循环联接”（它会将一个关系中的各个元组与另一个关系中的各元组一一配对）更高效。索引联接和排序联接的运行时间接近于查看所涉及的两个关系并生成联接结果所需的时间。
- 关系代数表达式的优化可以大大缩短表达式求值的运行时间，因此如果实践中用于表示查询的语言是基于关系代数的，这种优化是很关键的。
- 改善给定表达式运行时间的方式有很多种，将选择往下压通常是其中收益最大的。

8.11　参考文献

对数据库，特别是那些基于关系模型的数据库的进一步研究可以在Ullman [1988]中找到。

尽管不少更早的文献中也包含了一些这样的概念，但是Codd [1970]的论文通常被视为关系数据模型的起源。最早使用这一模型实现的系统是伯克利的INGRES（Stonebrake et al. [1976]）和IBM的System R（Astrahan et al. [1976]）。后者是8.7节中简述过的SQL语言的起源，而且至今仍出现在很多数据库管理系统中，见Chamberlin et al. [1976]。还可以在UNIX的awk命令中找到关系模型（Aho, Kernighan, and Weinberger [1988]）。

Aho, A. V., B.W. Kernighan, and P. J.Weinberger [1988]. *The AWK programming Language*, Addison-Wesley, Reading, Mass.

Astrahan, M. M. et al. [1976]. “System R: a relational approach to data management,” *ACM Trans. on Database Systems* 1:2, pp. 97–137.

Chamberlin, D. D. et al. [1976]. “SEQUEL 2: a unified approach to data definition, manipulation, and control,” *IBM J. Research and Development* **20**:6, pp. 560–575.

Codd, E. F. [1970]. “A relational model for large shared data banks,” *Comm. ACM* **13**:6, pp. 377–387.

Stonebraker, M., E. Wong, P. Kreps, and G. Held [1976]. “The design and implementation of INGRES,” *ACM Trans. on Database Systems* **1**:3, pp. 189–222.

Ullman, J. D. [1988]. *Principles of Database and Knowledge-Base Systems* (two volumes) Computer Science Press, New York.

第 9 章

图数据模型

从某种意义上讲，图就是二元关系。不过，它利用一系列由线（称为边）或箭头（称为弧）连接的点（称为节点）提供了强大的视觉效果。在这方面，图就是我们在第5章中研究过的树数据模型的泛化。和树一样，图也有多种形式：有向图/无向图，以及标号图/无标号图。

图还和树一样可以解决大范围的问题，比如距离的计算、关系中环的查找，以及连通性的确定。我们在第2章中已经见识过用图来表示程序的结构。而第7章也用到图来表示二元关系，并用图展示了关系的某些属性，比如交换律。在第10章中将看到用图表示自动机，而在第13章中会看到用图表示电路。而图除上述这些之外的若干重要应用将在本章中讨论。

9.1　本章主要内容

本章的主要内容包括以下这些。

- 与有向图和无向图有关的定义（9.2节和9.10节）。
- 表示图的两种重要数据结构：邻接表和邻接矩阵（9.3节）。
- 用来在无向图中找出连通分支的算法和数据结构（9.4节）。
- 找出最小生成树的技巧（9.5节）。
- 名为“深度优先搜索”的用于探索图的实用技巧（9.6节）。
- 应用深度优先搜索测试有向图是否有环路，找出无环图的拓扑次序，以及确定是否存在从一个节点到另一节点的路径（9.7节）。
- 用来找出最短路径的迪杰斯特拉算法（9.8节）。该算法可找出从某个“源”节点到每个节点的最小距离。
- 找出任意两个节点间最小距离的弗洛伊德算法（9.9节）。

本章中的很多算法都是比解决问题的直观方法更加高效的实用技巧。

9.2　基本概念

有向图，是由*节点*集合N以及N上的二元关系A组成的。我们将A称为有向图*弧*的集合，因此弧是节点的有序对。

绘出的图如图9-1所示。各节点是用圆圈表示的，节点的名称就在圆圈中央。我们通常会用从0开始的整数为节点命名，或者使用等效的枚举。在图9-1中，节点集合$N=\{0，1，2，3，4\}$。

A中的弧(u, v)都是由从u到v的箭头表示的。在图9-1中，弧的集合是

$$A = \{(0,0), (0,1), (0,2), (1,3), (2,0), (2,1), (2,4), (3,2), (3,4), (4,1)\}$$

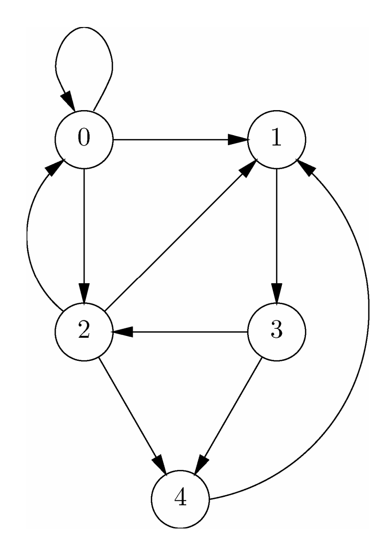


图9-1　有向图的示例

在文本中，一般习惯将弧(u, v)表示为$u \to v$。我们将v称为弧的*头部*，而将u称为弧的*尾部*，以适应v在箭头的头部而u在其尾部的概念。例如，$0 \to 1$就是图9-1中的一条弧，它的头部是节点1，而尾部是节点0。另一条弧是$0 \to 1$，这样一条从某节点通向其自身的弧就叫作*自环*（loop）。对该弧而言，头部和尾部都是节点0。

9.2.1　前导和后继

当$u \to v$是弧时，还可以说u是v的*前导*（predecessor），而且v是u的*后继*（successor）。因此，弧$0 \to 1$就表示0是1的前导而1是0的后继，而弧$0 \to 0$则表示0同时是其本身的前导和后继。

9.2.2　标号

就像对树那样，也可以为图的各节点附加*标号*（label）。标号是绘制在所对应的节点附近的。同样，可以在靠近弧中点的位置为弧放置标号。节点的标号或弧的标号可以是任意类型的。例如，图9-2就展示了节点名为1，标号为“狗”，节点名为2，标号为“猫”，而且有一条标号为“咬”的弧$1 \to 2$。

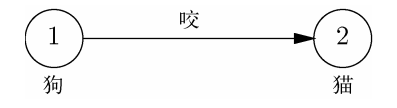


图9-2　含两个节点的标号图

和树一样，不应该把节点的名称与其标号弄混。同一幅图中各节点的名称必须是唯一的，但可能有不止一个节点的标号相同。

9.2.3　路径

有向图中的*路径*是一列节点(v_1, v_2, ⋯, v_k)，其中每个节点都有到下一节点的弧，也就是$v_i \to v_{i+1}$，$i = 1, 2, \cdots, k-1$。该路径的*长度*是$k-1$，也就是这条路径上弧的数量。例如，图9-1中的(0，1，3)就是一条长度为2的路径。

$k=1$的情况也是可以存在的。也就是说，任何节点v本身都是一条从v到v的长度为0的路径。该路径上没有弧。

9.2.4　有环图和无环图

有向图中的*环路*（cycle）是指起点和终点为同一节点的长度不为0的路径。*环路的长度*就是这条路径的长度。请注意，路径长度为0的情况不是环路，虽然“其起点和终点是同一节点”。然而，由一条弧$v \to v$构成的路径是一条长度为1的环路。

✦　示例 9.1

考虑图9-1中的图。因为有自环$0 \to 0$，存在一条长度为1的环路(0，0)。还有一条长度为2的环路(0, 2, 0)，因为有弧$0 \to 2$和$2 \to 0$。同样，(1, 3, 2, 1)是一条长度为3的环路，而(1, 3, 2, 4, 1)则是长度为4的环路。

请注意，环路的起点和终点可以是其中的任一节点。也就是说，环路(v_1, v_2, ⋯, v_k, v_1)也可以写为(v_2, ⋯, v_k, v_1, v_2)，或者写为(v_3, ⋯, v_k, v_1, v_2, v_3)，等等。例如，环路(1, 3, 2, 4, 1)也可以写为(2, 4, 1, 3, 2)。

在每条环路中，第一个节点和最后一个节点都是相同的。如果环路(v_1, v_2, ⋯, v_k, v_1)的节点v_1、⋯、v_k中没有一个出现一次以上，就说该环路是*简单环路*，也就是说，简单环路的唯一重复出现在最终节点处。

✦　示例 9.2

示例9.1中的环路都是简单环路。在图9-1中，环路(0，2，0)是简单环路。不过，也有些环路不是简单环路，比如环路(0, 2, 1, 3, 2, 0)中节点2就出现了两次。

给定含有节点v的非简单环路，就能找到含有v的简单环路。要知道原因，可以假设有一条起点和终点都是v的环路(v, v_1, v_2, ⋯, v_k, v_1)。如果该环路不是简单环路，就只会是以下两种情况之一。

(1) v出现了3次或3次以上；

(2) 存在某个v之外的节点u，它出现了两次，也就是，环路肯定是(v, ⋯, u, ⋯, u, ⋯, v)这样的。

在第(1)种情况下，可以直接删除倒数第二次出现v的位置之前的所有节点，结果是一条从v到v的更短环路。在第(2)种情况中，可以删除从u到u的部分，将其用一个u替代，得到环路(v, ⋯, u, ⋯, v)。不管哪种情况，得到的结果肯定仍然是环路，因为结果中的每条弧都是原环路中的，因此肯定是出现在该图中的。

在让环路成为简单环路之前，可能有必要多次重复该变形。因为环路在每次迭代后总会变得更短，所以最终一定能得到简单环路。我们刚刚已经证明了，如果图中有一条环路，那么一定至少含有一条简单环路。

✦ 示例 9.3

给定环路(0，2，1，3，2，0)，可以删除第一个2以及它后面的1和3，这样就得到了简单环路(0，2，0)。从实际情况上讲，该环路是从0开始，到达2，然后是1，再是3，回到2，最后回到0。第一次到达2时，可以假装这是第二次到达2，跳过了1和3，然后直接回到0。

再举个例子，考虑非简单环路(0，0，0)。因为0出现了3次，所以可以删除第一个0，也就是删除倒数第二个0之前的所有内容。实际上我们是将绕着自环 $0 \to 0$ 行进两次的路径替换为绕行一次的路径。

如果图中含一条或多条环路，就说该图是*有环的*。如果不含环路，就说该图是*无环的*。而根据刚才有关简单环路的论证，当且仅当图中含有简单环路时，该图为有环图，因为如果该图有环路，那么它肯定有简单环路。

✦ 示例 9.4

我们在3.8节中提到过，可以用名为"调用图"的有向图表示由一系列函数执行的调用。图中的节点就是函数，而如果函数p调用了函数Q，就有一条弧 $P \to Q$ 。例如，图9-3展示了与2.9节中的归并算法对应的调用图。

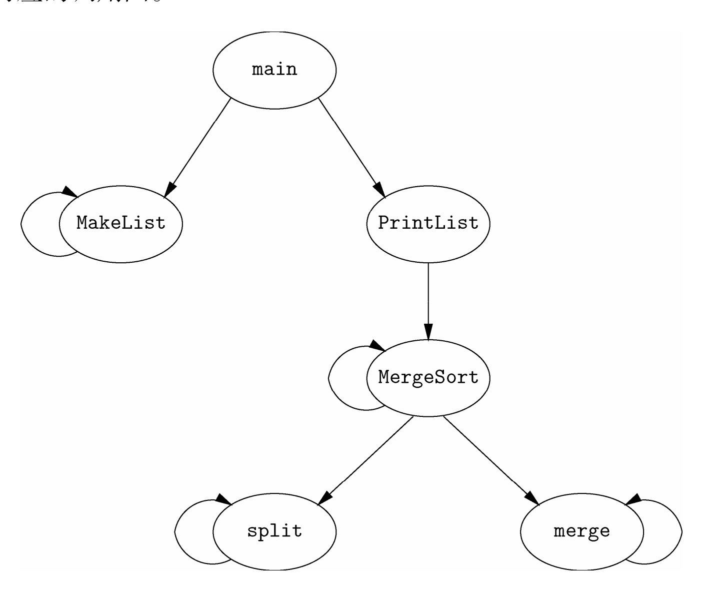
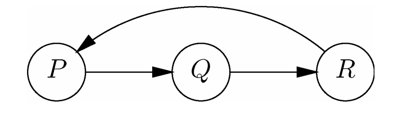


图9-3　归并排序算法的调用图

调用图中出现的环路意味着算法中的递归。在图9-3中有4条简单环路，分别是围绕节点`MakeList`、`MergeSort`、`split`和`merge`的长度为1的环路。而且每条环路都是自环。回想一下，这些函数都调用了自己，因此都是递归的。到目前为止，函数调用自己的递归是最常见的类型，而且这些递归在调用图中都是以自环的形式出现的。我们将这种递归称为*直接递归*。不过，大家偶然会看到*间接递归*，也就是调用图中环路长度大于1的递归。例如，下图就表示函数P调用了函数Q，而函数Q调用了函数R，函数R回过头来又调用了函数P。



9.2.5 无环路径

如果路径中没有节点出现一次以上，就说该路径是*无环的*。我们刚才在证明对每条环路而言都存在一条简单环路时给出的论证也说明了以下原则。如果存在从u到v的路径，就存在从u到v的无环路径。要知道原因，首先看看从u到v的任意路径。如果存在某个节点w（它可能是u或v）出现了两次，就可以将两个w以及这两个w之间的所有内容用一个w替代。就像环路的情况那样，我们可能不得不多次重复该过程，但最终会将该路径简化为无环路径。

✦ 示例 9.5

再次考虑图9-1中的图。路径(0，1，3，2，1，3，4)是从0到4的包含了环路的路径。我们可以将注意力放在两个节点1上，并将它们及其之间的3和2替换为1，留下(0，1，3，4)，这是条无环路径，因为没有节点出现了两次。将注意力放在两个节点3上也可以得到同样结果。

9.2.6 无向图

有时候也可以用没有方向的线条（称为*边*）连接节点。正式地讲，边是两个节点组成的集合。边$\{u, v\}$表示节点u和v是双向连通的。[①]如果$\{u, v\}$是边，那么节点u和v就是*邻接的*或者说是*邻居*。带有边的图，也就是有着对称弧关系的图，就称为*无向图*。

✦ 示例 9.6

图9-4表示了夏威夷群岛的部分公路图。城市之间的公路就是用边表示的，而且边的标号表示的是行车距离。将公路表示为边而不是弧是很自然的，因为公路通常是双向的。

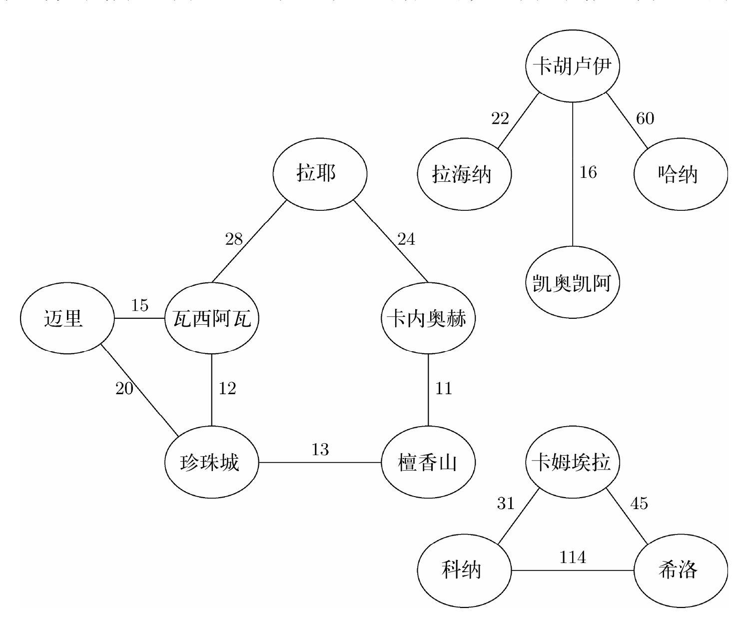


图9-4 表示夏威夷群岛中瓦胡岛、毛伊岛和夏威夷岛（左起顺时针排列）上的公路的无向图

① 请注意，边必须刚好有两个节点。由一个节点构成的单一集不是边。因此，虽然从一个节点到其自身的边是可以存在的，但从一个节点到其自身的自环边是不能存在的。不过某些“无向图”的定义也允许这种自环存在。

9.2.7　无向图中的路径和环路

无向图的路径是各节点与下一节点间都由边连通的节点列($v_1, v_2, \cdots, v_k$)。也就是说，对$i=1, 2, \cdots, k-1$而言，$\{v_i, v_{i+1}\}$是边。请注意，边作为集合，其中的元素是没有特定次序的。因此，边$\{v_i, v_{i+1}\}$也可以表示为$\{v_{i+1}, v_i\}$。

路径($v_1, v_2, \cdots, v_k$)的长度是$k-1$。就像有向图那样，节点本身是一条长度为0的路径。

在无向图中定义环路是有点棘手的。问题在于，我们不希望将诸如(u, v, u)这样只要存在边$\{u, v\}$就存在的路径视作环路。同样，如果($v_1, v_2, \cdots, v_k$)是条路径，我们可以来回穿越该路径，但肯定不想把如下路径视为环路。

$$(v_1, v_2, \cdots, v_{k-1}, v_k, v_{k-1}, \cdots, v_2, v_1)$$

在无向图中定义简单环路的最简单方式可能是指长度不小于3而且起点和终点为同一节点，而且预期最后的节点不会与任何节点重复的路径。而无向图中非简单环路的概念通常不怎么使用，所以后面就不继续讲这个概念了。

和有向环路一样，如果两条无向环路是由次序相同的相同节点构成，就可以将它们视作相同环路。如果两条无向环路是由次序相反的相同节点构成，那么它们也是相同的环路。正式地讲，对从1到k的每个i，简单环路($v_1, v_2, \cdots, v_k$)与环路($v_i, v_{i+1}, \cdots, v_k, v_1, v_2, \cdots, v_{i-1}$)及环路($v_i, v_{i-1}, \cdots, v_1, v_k, v_{k-1}, \cdots, v_{i+1}$)都是等价的。

✦　示例 9.7

在图9-4中，(瓦西阿瓦，珍珠城，迈里，瓦西阿瓦) 是长度为3的简单环路。而如果从迈里开始并沿着同样的次序行经环路，它也可以写为等价的(迈里，瓦西阿瓦，珍珠城，迈里)。同样，也可以从珍珠城开始，并沿着相反方向行经环路，得到等价的环路(珍珠城，迈里，瓦西阿瓦，珍珠城)。

再举个例子，(拉耶，瓦西阿瓦，珍珠城，檀香山，卡内奥赫，拉耶)是一条长度为5的简单环路。

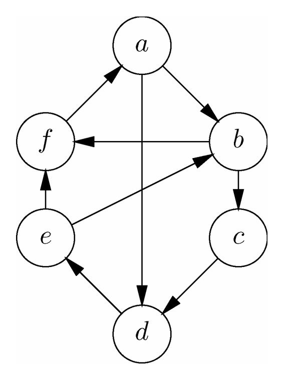


图9-5　习题(1)和习题(2)对应的有向图

9.2.8　习题

(1) 考虑图9-5中的图。

(a) 总共有多少条弧？

(b) 从节点a到节点d共有多少条无环路径？它们分别是什么？

(c) 节点b的前导是什么？

(d) 节点b的后继是什么？

(e) 总共有多少条简单环路？列出它们。不要重复只有起点不同的路径（见习题(8)）。

(f) 列出长度最多到7的所有非简单环路。

(2) 通过将弧$u \to v$替换为边$\{u, v\}$，把图9-5所示的图转换为无向图。

(a) 找出从a到d的所有无重复节点的路径。

(b) 包含全部6个节点的简单环路有几条？列出这些环路。

(c) 节点a的邻居有哪些？

(3) * 如果某图有10个节点，那么它最多可以有多少条弧？它最少可能有多少条弧？一般来说，如果图有n个节点，那么它最多和最少分别可能含有多少条弧？

(4) * 针对无向图的边重复习题(3)。

(5) ** 如果某有向图是无环的而且有n个节点，那么它最多可能有多少条弧？

(6) 在目前为止本书介绍过的内容中找出一个函数间间接递归的例子。

(7) 以所有可能的方式写出环路(0，1，2，0)。

(8) * 设G是有向图，并设R是G的环路上的关系，当且仅当$(u_1, \cdots, u_k, u_1)$和$(v_1, \cdots, v_k, v_1)$表示相同环路时有$(u_1, \cdots, u_k, u_1)\ R\ (v_1, \cdots, v_k, v_1)$。证明$R$是$G$的环路上的等价关系。

(9) * 如果S是定义在某图节点上的关系，当且仅当$u = v$或者存在同时包含节点u和v的环路时有uSv，证明关系S是该图节点上的等价关系。

(10) * 当讨论无向图中的简单环路时，我们提到过如果两条环路有着相同的节点，不管是次序相同还是次序相反，这两条环路其实都是相同的。证明：由表示相同简单环路的有序对组成的关系R是等价关系。

9.3 图的实现

实现图的标准方式有两种。一种叫作*邻接表*，大致上与二元关系的实现方法类似。第二种叫作*邻接矩阵*，是一种表示二元关系的新方法，而且更适合表示那些有序对数量占据了可能在某给定定义域中浮动的有序对总数很大一部分的关系。我们将首先为有向图考虑这些表示，然后再为无向图考虑。

9.3.1 邻接表

设节点是由整数0、1、…、MAX-1或者等价的枚举类型命名的。一般而言，我们会使用`NODE`作为节点的类型，不过可以假设`NODE`跟`int`是一回事。那么就可以使用7.9节介绍的一般化的特征向量法表示弧的集合，这种表示就叫*邻接表*。我们将链表的节点定义如下：

```
typedef struct CELL *LIST;
struct CELL {
    NODE nodeName;
    LIST next;
};
```

然后创建数组：

```
LIST successors[MAX];
```

也就是说，`successor[u]`这一项包含了一个指针，指向由节点u的所有后继组成的链表。

✦ 示例 9.8

图9-1中的图可以用图9-6中的邻接表表示。我们已经通过节点编号为这些邻接表排过序了，不过节点的后继可能以任意次序出现在其邻接表中。

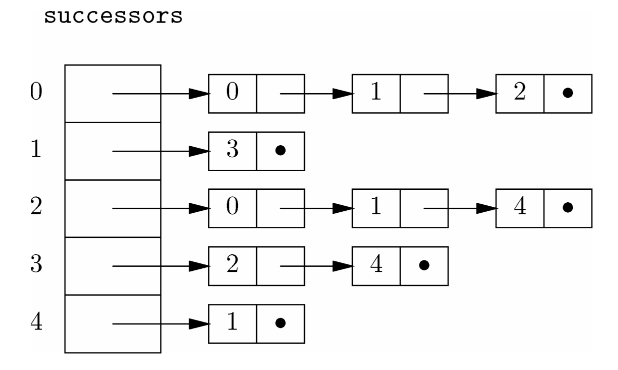


图9-6　图9-1中的图的邻接表表示

9.3.2　邻接矩阵

另一种常见的有向图表示方式是邻接矩阵。我们可以创建如下二维数组：

```
BOOLEAN arcs[MAX][MAX];
```

其中如果存在弧 $u \to v$ ，则arcs[u][v]的值为TRUE，否则该值为FALSE。

✦ 示例 9.9

图9-1中的图对应的邻接矩阵如图9-7所示。用1表示TRUE，用0表示FALSE。

	0	1	2	3	4
0	1	1	1	0	0
1	0	0	0	1	0
2	1	1	0	0	1
3	0	0	1	0	1
4	0	1	0	0	0

图9-7　表示图9-1中的图的邻接矩阵

9.3.3　对图的操作

如果考虑一些简单的图操作，就可以看到图的这两种表示方法间的不同。最基本的操作也许就是确定从节点u到节点v是否存在弧 $u \to v$ 。在邻接矩阵中，要查找arcs[u][v]看该项是否为TRUE只需要$O(1)$的时间。

邻接矩阵与邻接表的对比

当图很稠密时，也就是，当弧的数量接近最大可能数字时（对有n个节点的图来说是n^2），就倾向于选择邻接矩阵表示。不过，如果图是稀疏的，也就是说，如果可能存在的弧大多数并

未出现，那么用邻接表表示法就可能节省空间。要知道原因，请注意表示含n个节点的图的邻接矩阵有n^2位，假设用1位来表示TRUE和FALSE，而不是像本节中已经做过的那样用整数表示。

在一般的计算机中，像邻接表单元这样的结构体是由整数和指针构成的，每个结构体会使用32位表示整数，并用32位表示指针，总共需要64位。因此，如果弧的数量为a，就需要$64a$位存储该链表，而且存放n个表头的数组还需要$32n$位。如果有$32n + 64a<n^2$，也就是如果有$a<n^2/64-n/2$，邻接表就要比邻接矩阵节省空间。如果n很大的话，就可以舍掉$n/2$这项，并近似地将之前的不等式视作$a<n^2/64$，也就是说，如果可能存在的弧只有不到1/64实际出现，邻接表就要比邻接矩阵节省空间。我们在讨论对图的操作时将更为详细地论证这两种表示法的优劣。下表总结了为多种操作优先选择的表示法。

操　作	稠　密　图	稀疏图
查找弧	邻接矩阵	皆可
找后继	皆可	邻接表
找前导	邻接矩阵	皆可

使用邻接表的话，就要找到对应u的邻接表的表头，需要$O(1)$的时间。然后，如果v不在表中，就要遍历该表直到末端，或者如果v存在的话平均要浏览该表的一半。如果该图中有a条弧和n个节点，那么平均要花$O(1+a/n)$的时间来完成这样的查找。如果a不大于n乘以某个常数因子，这个量就是$O(1)$。不过，在与n相比时，a越大，使用邻接表表示法验证弧是否出现所花的时间就越长。在a大约是n^2（其最大可能值）的极端情况下，每个邻接表中都将近有n个节点。这种情况下，找到某给定的弧平均要花$O(n)$的时间。换句话说，当我们需要查找某给定的弧时，图越稠密，就越愿意选择邻接矩阵而不是邻接表。

另一方面，我们经常需要找到某给定节点u的所有后继。要用邻接表找到所有的后继，就要行向`successor[u]`并遍历该表，平均耗时$O(a/n)$。如果a和n是近似的，就可以在$O(1)$的时间内找到u的所有后继。不过如果使用邻接矩阵，就必须检查节点u所在的那一整行，不管a是多少都要花$O(n)$的时间。因此，对每个节点只有少量边与之连接的图来说，在需要检查某给定节点的后继时，使用邻接表要比使用邻接矩阵快得多。

不过，假设想要找到某给定节点v的全部前导。如果用邻接矩阵表示，就需要检查v所在的那一整列，如果u所在那行的位置是1，就意味着u是v的前导。这一检查要花费$O(n)$的时间。而邻接表表示在查找前导时也帮不上忙。必须检查对应每个节点u的邻接表，看看该表中是否含有v。因此，我们可能要检查所有邻接表的所有单元，而且很可能将检查大多数的单元。因为整个邻接表结构中单元的数量等于a，也就是图中弧的数量，所以使用邻接表在含a条弧的图中找前导的时间就是$O(a)$。在这里，邻接矩阵表示法是占优势的，而且图越稠密，这种优势就越大。

度的问题

从节点v出发的弧的数量就叫作v的出度。因此，节点的出度等于其邻接表的长度，还等于相应的邻接矩阵中对应v的那行中1的数量。进入节点v的弧的数量叫作v的入度。入度衡量的是节点v在某节点的邻接表中出现的次数，而且是相应的邻接矩阵中对应v的那列中1的数量。

在无向图中，我们不会区分边是从节点出发还是进入节点。对无向图而言，节点v的度就

是v的邻居的数量，也就是含有v的边$\{u, v\}$的数量。请记住，在集合中，成员的次序是不重要的，所以$\{u, v\}$和$\{v, u\}$是相同的边，因此只能计算一次。而无向图的度则是该图中节点的度的最大值。例如，如果将二元关系看作无向图，那么它的度就是3，因为节点最多只能与它的父节点、左子节点和右子节点之间有边。对有向图而言，可以说有向图的入度是其节点入度的最大值，同样，有向图的出度是其节点出度的最大值。

9.3.4 无向图的实现

如果图是无向图，可以假装每条边都被替代为两个方向上的弧，并将得到的有向图用邻接表或邻接矩阵表示出来。如果使用邻接矩阵，那么该矩阵是对称的。也就是说，如果称该矩阵为edges，那么$edges[u][v] = edges[v][u]$。如果使用邻接表表示法，那么边$\{u, v\}$就会被表示两次。我们可以在邻接表中找到对应$u$的$v$，也可以在该表中找到对应$v$的$u$。这种排列通常是实用的，因为不可能事先分辨出$\{u, v\}$这条边是更可能从$u$到$v$，还是更可能从$v$到$u$。

✦ 示例 9.10

考虑一下如何表示图9-4的无向图中最大那部分，也就是表示瓦胡岛6个城市的那部分。在这里我们要忽略边的标号。对应的邻接矩阵表示就如图9-8所示。请注意，该矩阵是对称的。

	拉耶	卡内奥赫	檀香山	珍珠城	迈里	瓦西阿瓦
拉耶	0	1	0	0	0	1
卡内奥赫	1	0	1	0	0	0
檀香山	0	1	0	1	0	0
珍珠城	0	0	1	0	1	1
迈里	0	0	0	1	0	1
瓦西阿瓦	1	0	0	1	1	0

图9-8 图9-4中的无向图的邻接矩阵表示

图9-9展示了该无向图的邻接表表示。在两种情况下，我们都要用到枚举类型

```
enum CITYTYPE {Laie, Kaneohe, Honolulu,
    PearlCity, Maili, Wahiawa};
```

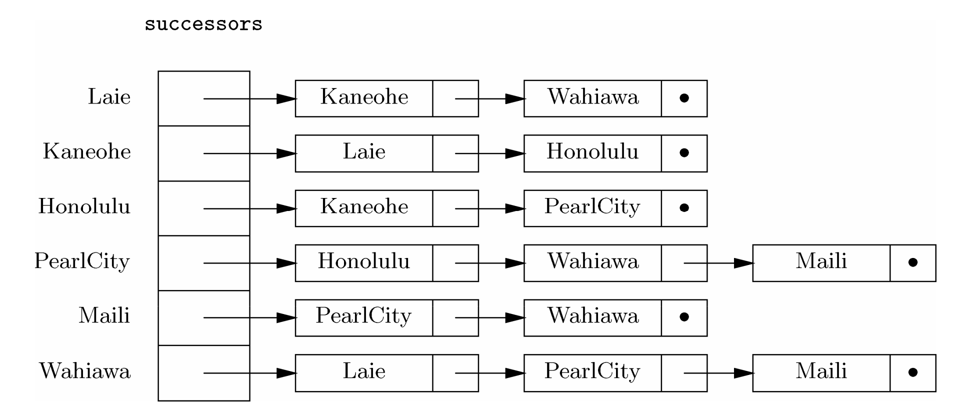


图9-9 图9-4中无向图的邻接表表示

来作为数组的索引。这种安排多少有些刻板，因为这样定义就没法对该图的节点集合进行任何改动。不过我们很快就会给出用整数显式命名节点并用城市名作为节点标号的相似示例，这样在改变节点集合方面会更具灵活性。

9.3.5　标号图的表示

假设图的弧（如果是无向图就是边）带有标号。在使用邻接矩阵时，就可以将表示弧 $u \to v$ 在图中出现的1替换为该弧的标号。而且必须要有一些可作为矩阵的项但又不会与标号混淆的值，我们要用该值表示弧未出现的情况。

如果用邻接表表示图，就要为构成各链表的单元添加一个`nodeLabel`字段。如果存在标号为L的弧 $u \to v$，那么在对应节点u的邻接表中就会找到`nodeName`字段为v而且`nodeLabel`字段为L的单元。`nodeLabel`字段的值就表示该弧的标号。

我们要用另一种方式表示节点的标号。对邻接矩阵来说，只要创建另一个名为`NodeLabels`的数组，并设`NodeLabels[u]`是节点u的标号。在使用邻接表时，已经有了以节点为索引的表头数组。我们要把该数组的元素改为结构体，一个字段为节点标号，而另一个字段为指向邻接表开头的指针。

✦　示例 9.11

我们要再次表示图9-4所示图的较大部分，不过这次要加上边的标号，也就是距离。此外，要给出节点的整数名称，从对应拉耶的0开始，按照顺时针方向排列。城市的名称是用节点的标号表示的。要将节点标号的类型定义为长度为32的字符数组。这种表示方式要比示例9.10的方式更灵活，因为如果要在数组中分配额外的位置，就可以在想要添加城市时如愿以偿。得到的图重绘为图9-10的样子，而对应的邻接矩阵表示如图9-11所示。

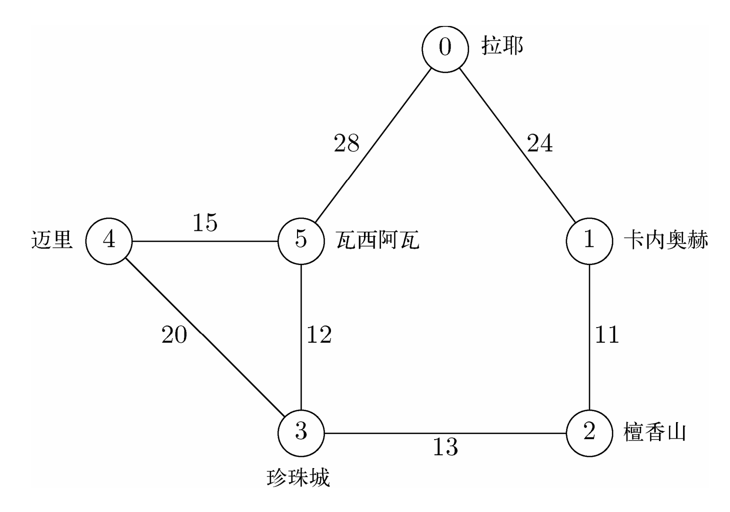


图9-10　节点名称为整数而标号为城市名的瓦胡岛地图

请注意，这一表示其实有两个部分，`cities`数组，指示从0到5的整数代表的城市，以及`distances`矩阵，指示边是否出现以及出现的边的标号。我们用不会被误解为标号的值−1表示未出现的边，因为在本例中，标号是表示城市间距离的，它肯定是正数。

cities

0	拉耶
1	卡内奥赫
2	檀香山
3	珍珠城
4	迈里
5	瓦西阿瓦

distances

	0	1	2	3	4	5
0	−1	24	−1	−1	−1	28
1	24	−1	11	−1	−1	−1
2	−1	11	−1	13	−1	−1
3	−1	−1	13	−1	20	12
4	−1	−1	−1	20	−1	15
5	28	−1	−1	12	15	−1

图9-11　有向图的邻接矩阵表示

可以将该结构体声明如下：

```
typedef char CITYTYPE[32];
typedef CITYTYPE cities[MAX];
int distances[MAX][MAX];
```

这里的*MAX*是至少为6的某个数字，它限制了可以出现在图中的节点数。`CITYTYPE`被定义为长度为32的字符数组，而且数组`cities`给出了各节点的标号。例如，可以预期*cities*[0]是`"Laie"`（拉耶）。

还可以用邻接表表示图9-10所示的图。假设常量*MAX*和`CITYTYPE`类型都与上面的定义相同。我们可以将`CELL`和`LIST`类型定义为

```
typedef struct CELL *LIST;
struct CELL {
    NODE nodeName;
    int distance;
    LIST next;
};
```

接下来，要将`cities`数组声明为

```
struct {
    CITYTYPE city;
    LIST adjacent;
} cities[MAX];
```

图9-12展示了用这种方式表示的图9-10所示的图。

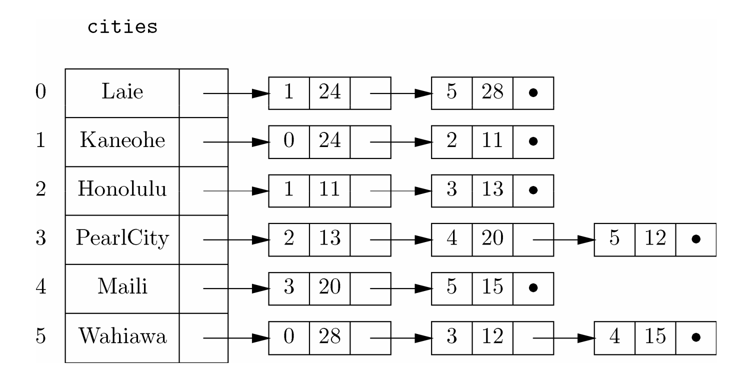


图9-12　具有节点标号和边标号的图的邻接表表示

9.3.6　习题

(1) 分别用

(a) 邻接表

(b) 邻接矩阵

表示图9-5（见9.2节的习题）所示的图，在每种情况下都给出合适的类型定义。

(2) 假设图9-5中的弧都是边，即将图变成无向图。针对该无向图重复习题(1)。

(3) 为图9-5中有向图的弧加上标号，标号是由弧的尾部跟上弧的头部组成的长度为2的字符串。例如，弧 $a \to b$ 的标号就是字符串`ab`。还有，假设节点的标号是对应其名称的大写字母。比如名为a的节点的标号就是`A`。针对该带标号的有向图重复习题(1)。

(4) * 无标号图的邻接矩阵表示与弧集合的特征向量表示有何关系？

(5) * 通过对n的归纳证明，在含n个节点的无向图中，节点度的和是边数的两倍。注意。不使用归纳法也是可以证明该命题的，不过这里要求大家使用归纳法证明。

(6) 设计算法，在有向图的(a)邻接矩阵；(b)邻接表表示中插入和删除弧。

(7) 针对无向图重复习题(6)。

(8) 我们可以为有向图或无向图的邻接表表示增加“前导表”。在执行以下哪些操作时，会选择这种表示？

(a) 查找弧。

(b) 找出所有后继。

(c) 找出所有前导。

在分析中要同时考虑稠密图和稀疏图的情况。

9.4　无向图的连通分支

我们可以将任意无向图分解为一个或多个连通分支（connected component）。连通分支是节点的集合，分支的任意成员之间都是存在路径的。此外，连通分支是极大的，也就是说，连通分支中的节点没有与分支外的任意节点连接的路径。如果图是由单个连通分支组成的，那么就说该图是连通图。

连通分支的物理解释

如果给定了绘制出的无向图，就很容易看出连通分支。将边想象成弦。如果拿起任一节点，包含该节点作为成员的连通分支就会随之而来，而其他连通分支中的成员则会待在原地。当然，这些“眼球”很容易完成的任务让计算机完成起来却不一定很容易。找出图中连通分支的算法是本节的重要主题。

✦ 示例 9.12

再次考虑图9-4中有关夏威夷群岛的图。其中含有3个连通分支，分别对应3个岛。最大的分支是由拉耶（Laie）、卡内奥赫（Kaneohe）、檀香山（Honolulu）、珍珠城（Pearl City）、迈里（Maili）和瓦西阿瓦（Wahiawa）组成的。这些城市都是在瓦胡岛上，而且它们显然是由公路（也就是边的路径）相互连接。还有，瓦胡岛上的公路是没法连接到其他岛的。按照图论的说法就是，在图9-4中，不存在从上面提到的这6个城市到其他城市的路径。

第二个分支是由毛伊岛上的城市拉海纳（Lahaina）、卡胡卢伊（Kahului）、哈纳（Hana）和凯奥凯阿（Keokea）组成的。第三个分支是夏威夷“大岛”上的城市希洛（Hilo）、科纳（Kona）和卡姆埃拉（Kamuela）。

9.4.1 作为等价类的连通分支

另一种看待连通分支的实用方式就是将其视为等价关系P上的等价类，其中P是定义在无向图节点上的关系，当且仅当存在从u到v的路径时有uPv。很容易验证P是等价关系。

(1) P是自反的，也就是说，对任意节点u有uPu，因为从任意节点到其自身都有长度为0的路径。

(2) P是对称的。如果uPv，那么存在从u到v的路径。因为该图是无向图，所以相反的节点序列也是路径。因此有vPu。

(3) P是传递的。假设uPw和wPv都成立，那么存在从u到w的路径，比方说是

$$(x_1, x_2, \cdots, x_j)$$

这里有$u = x_1$且$w = x_j$。还有，存在从w到v的路径$(y_1, y_2, \cdots, y_j)$，其中$w = y_1$而且$v = y_j$。如果将这些路径连接在一起，就得到了从u到v的路径，即

$$(u = x_1, x_2, \cdots, x_j{=}w{=}y_1, y_2, \cdots, y_k = v)$$

✦ 示例 9.13

考虑图9-10中从檀香山到迈里的路径(檀香山，珍珠城，瓦西阿瓦，迈里)。再考虑该图中从迈里到拉耶的路径(迈里，珍珠城，瓦西阿瓦，拉耶)。如果将这两条路径连在一起，就得到了从檀香山到拉耶的路径：

(檀香山，珍珠城，瓦西阿瓦，迈里，珍珠城，瓦西阿瓦，拉耶)

这条路径刚好是条环路。正如在9.2节中提过的，我们总是能删除环路得到无环路径。在这种情况下，要消除环路，一种方法就是将两个瓦西阿瓦以及它们之间的节点用一个瓦西阿瓦替代，得到从檀香山到拉耶的无环路径

(檀香山，珍珠城，瓦西阿瓦，拉耶)

因为P是等价关系，所以它把问题中无向图节点的集合分成了等价类。包含节点v的类就是满足vPu的所有节点u的集合。此外，等价类的其他属性还有，如果节点u和v在不同的等价类中，那么不可能有uPv,也就是说不存在从一个等价类中的某节点到另一等价类中节点的路径。因此，由“路径”关系P定义的各等价类就是该图的各连通分支。

9.4.2　计算连通分支的算法

假设我们想构建图G的连通分支。一种方式是从G中没有边的节点组成的图G_0开始。然后考虑G的边，一次一条，构建一系列的图G_0、G_1、…，其中G_i是由G的节点和G的前i条边构成的。

依据。G_0是由G中没有边的节点组成。每个节点本身是一个分支。

归纳。假设在考虑了前i条边后得到了图G_i的连通分支，现在考虑第i+1条边：$\{u, v\}$。

(1) 如果u和v在G_i的同一分支中，那么G_{i+1}有着与G_i相同的连通分支集合，因为这条新的边不会连接到任何尚未连通的节点。

(2) 如果u和v在不同分支中，我们可以合并包含u和v的分支，得到G_{i+1}的连通分支。图9-13解释了为什么存在从u所在分支中任一节点x到v所在分支中任一节点y的路径。我们沿着第一个分支中从x到u的路径走，然后到边$\{u, v\}$，最后经过已知存在于第二个分支中的从v到y的路径。

当以这种方式考虑过所有的边时，就得到了全图的连通分支。

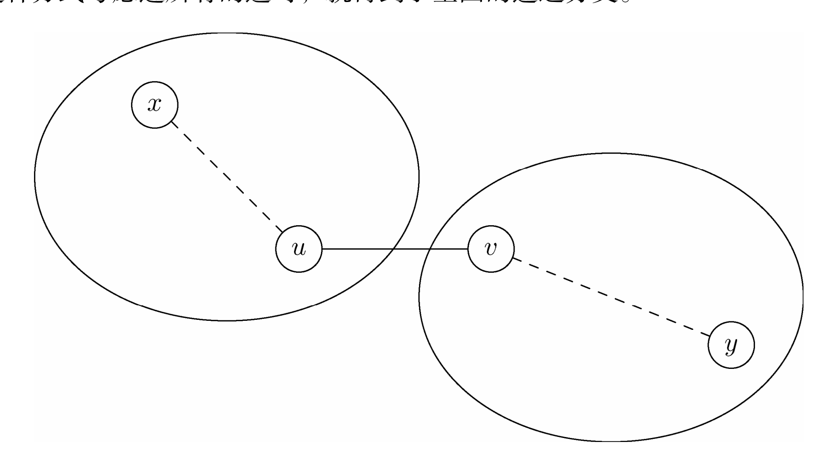


图9-13　添加连接了含u的分支与含v的分支的边$\{u, v\}$

✦ 示例 9.14

来考虑一下图9-4中的图。虽然我们能够以任意次序考虑这些边，不过为了9.5节中某一算法的需要，在这里按照边标号从小到大的次序列出了这些边。边的列表如图9-14所示。

首先，所有13个节点都在它们各自的分支中。当考虑1号边{卡内奥赫，檀香山}时，我们就将这两个节点合并到了一个分支中。而第二条边{瓦西阿瓦，珍珠城}则合并了这两个城市。第三条边是{珍珠城，檀香山}，这条边把包含这两个城市的分支合并了。至此，这些分支各含两个城市，所以就有了具有4个城市的分支，即{瓦西阿瓦，珍珠城，檀香山，卡内奥赫}。而所有其他城市都还在它们自己的分支中。

边	城市1	城市2	距　离
1	卡内奥赫	檀香山	11
2	瓦西阿瓦	珍珠城	12
3	珍珠城	檀香山	13
4	瓦西阿瓦	迈里	15
5	卡胡卢伊	凯奥凯阿	16
6	迈里	珍珠城	20
7	拉海纳	卡胡卢伊	22
8	拉耶	卡内奥赫	24
9	拉耶	瓦西阿瓦	28
10	科纳	卡姆埃拉	31
11	卡姆埃拉	希洛	45
12	卡胡卢伊	哈纳	60
13	科纳	希洛	114

图9-14　以标号为次序的图9-4中的边

4号边是{迈里，瓦西阿瓦}，并且将迈里添加到大分支中。第5条边是{卡胡卢伊，凯奥凯阿}，它将这两个城市合并到一条分支中。当考虑6号边{迈里，珍珠城}时，我们看到了新现象：这条边的两端已经存在于相同的分支中。因此我们不再合并6号边。

7号边是{拉海纳，卡胡卢伊}，它将节点拉海纳添加到了分支{卡胡卢伊，凯奥凯阿}上，形成了分支{拉海纳，卡胡卢伊，凯奥凯阿}。而8号边则将拉耶添加到了最大的分支上，最大分支现在就成了：

{拉耶，卡内奥赫，檀香山，珍珠城，瓦西阿瓦，迈里}

第九条边{拉耶，瓦西阿瓦}连接了该分支中的两个城市，因此被忽略掉。

10号边将卡姆埃拉和科纳组成一条分支，而且11号边为这一分支添加了希洛。12号边则将哈纳添加到了{拉海纳，卡胡卢伊，凯奥凯阿}这一分支中。最后，13号边{希洛，科纳}连接的是已经存在于同一分支中的两个城市。因此，最后总共有如下几个连通分支。

{拉耶，卡内奥赫，檀香山，珍珠城，瓦西阿瓦，迈里}

{拉海纳，卡胡卢伊，凯奥凯阿，哈纳}

{卡姆埃拉，希洛，科纳}

9.4.3　用于形成分支的数据结构

如果非正式地考虑9.4.2节中描述的算法，我们需要能迅速完成以下两项工作：

(1) 给定某节点，找出其当前所在分支；

(2) 将两个分支合并为一个分支。

有很多种数据结构可以支持这些操作。我们将研究一种简单却又能带来极佳性能的想法。关键在于将每个分支中的节点都放进一棵树中。[①]分支是由树的根表示的。上述两项操作现在可以按照如下方式实现。

---

① 请务必理解，在接下来的内容中，“树”和“图”指的是不同的结构。图的节点与树的节点间存在一一对应，也就是说，每个树节点都表示一个图节点。不过，树中父子节点之间的边并不一定是图中存在的边。

(1) 要找到图中节点的分支，就要找到该节点在树中的代表，然后沿着树中的路径到达表示该分支的根节点。

(2) 要合并两个不同的分支，我们就要让一个分支的根节点成为另一分支根节点的子节点。

✦ 示例 9.15

我们遵循示例9.14介绍的步骤，展示按照特定步骤创建的树。首先，每个节点本身都是一棵单节点树。而第一条边{卡内奥赫，檀香山}会让我们把两棵单节点树{卡内奥赫}和{檀香山}合并为一棵双节点树{卡内奥赫，檀香山}。任何一个节点都可以作为另一个的子节点。不过在这里假设檀香山是根节点卡内奥赫的子节点。

同样，第二条边{瓦西阿瓦，珍珠城}合并了两棵单节点树，而且可以假设珍珠城是根节点瓦西阿瓦的子节点。至此，当前的分支集合可以用图9-15所示的两棵树以及9棵单节点树来表示。

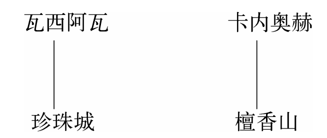


图9-15 合并分支得到的前两棵重要的树

第三条边{珍珠城，檀香山}合并了这两个分支。假设瓦西阿瓦是另一个根节点卡内奥赫的子节点。那么得到的分支就可以用图9-16中的树表示。

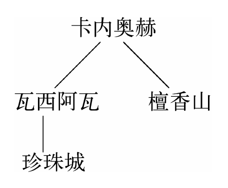


图9-16 表示含4个节点的分支的树

当考虑第四条边{瓦西阿瓦，迈里}时，就要把迈里合并到用图9-16中的树表示的分支中。既可以把迈里作为卡内奥赫的子节点，也可以将卡内奥赫当作迈里的子节点。不过这里选择前者，因为这样可以让树的高度保持比较小的值，而让大分支的根节点作为小分支根节点的子节点会让树中的路径变得更长。在确定节点的分支时，长路径会让我们要花更多时间才能沿着路径到达根节点。通过遵循这样的策略，并在分支高度相同时作出任意觉得，我们可能得到图9-17中表示3条最终连通分支的3棵树。

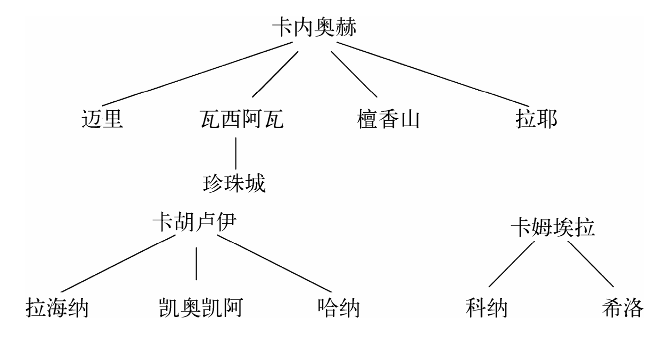


图9-17 使用树合并算法表示最终连通分支的树

遵循示例9.15中的经验，我们制订了这样的策略，在合并两棵树时，高度较小的根节点要成为高度较大的根节点的子节点。如果出现高度相等的情况就任选一种方式。从这一策略中得到的重要收获就是树的高度只能以树中节点数对数的速度增长，而且在实践中，树的高度往往还更小。因此，当沿着一条路径从树的节点到达其根节点时，所花的时间最多与树中节点数的对数成比例。我们可以通过对高度h的归纳证明如下命题，从而得出这一对数边界。

命题$S(h)$。按照将较低高度合并到较高高度的策略形成的高度h的树，至少有2^h个节点。

依据。依据是h=0。这样的树肯定只有一个节点，而且因为$2^0=1$，所以命题$S(0)$成立。

归纳。假设$S(h)$对某个$h \geqslant 0$成立，并考虑高度为$h+1$的树T。在通过合并形成T的过程中的某个时刻，树的高度第一次达到$h+1$。让树的高度达到$h+1$的唯一方式就是让某高度为h的树T_1的根节点成为某树T_2根节点的子节点。T是T_1加上T_2，可能还要加上一些后来要加上的其他节点，如图9-18所示。

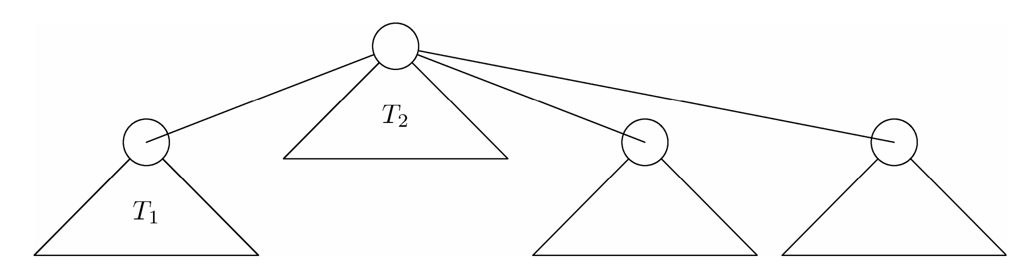


图9-18　形成高度为h+1的树

现在，根据归纳假设，T_1至少有2^h个节点。因为它的根节点成为了T_2根节点的子节点，所以T_2的高度也至少是h。因此，T_2也至少有2^h个节点。T是T_1加上T_2，可能还有更多节点组成的，所以T至少有$2^h+2^h=2^{h+1}$个节点。这就是$S(h+1)$，所以我们证明了归纳步骤。

现在就知道了，如果一棵树有n个节点且高度为h，那么肯定有$n \geqslant 2^h$。如果在两边取对数，就得到$\log_2 n \geqslant h$，也就是说，树的高度不可能大于节点数的对数。这样一来，当我们沿着任意路径从节点到达树的根节点时，都要花$O(\log n)$的时间。

现在要更详细地描述实现这些想法的数据结构。首先，假设用NODE类型来表示节点。就像以前那样，我们假设NODE的类型为int，而且MAX至少是图中所含节点的数量。对图9-4中的例子而言，要设MAX为13。

还要假设由EDGE类型的单元组成的链表edges，这些单元是由如下声明定义的

```
typedef struct EDGE *EDGELIST;
struct EDGE {
    NODE node1, node2;
    EDGELIST next;
};
```

最后，对图中的每个节点而言，还需要一个与之对应的树节点。树节点是TREENODE类型的结构体，由以下内容组成。

(1) 父指针，让我们能在该图的节点上构建树，并沿着树到达其根节点。父指针为NULL就标识该节点为根节点。

(2) 以给定节点为根节点的树的高度。只有该节点是根节点时才使用该高度。

因此可以将TREENODE类型定义为：

```
typedef struct TREENODE *TREE;
struct TREENODE {
    int height;
    TREE parent;
}:
```

我们还要定义数组

```
TREE nodes[MAX];
```

以便将每个图节点与树中某个节点关联起来。应当明白，数组nodes中的每一项都是指向树中节点的指针，而该数据项也是图中节点的唯一代表。

图9-19展示了两个重要的辅助函数。第一个是find，它接受节点a，取指向其对应树节点x的指针，沿着x的父指针及其祖先向上，直到到达根节点。这种对根节点的搜索是由第(2)行和第(3)行执行的。如果找到了根节点，就会在第(4)行返回指向该根节点的指针。请注意，在第(1)行，NODE类型一定是int，这样才能用它来作为nodes数组的索引。

```
        /* 返回树的根节点位置，该位置含有对应
           图节点a的树节点x */
        TREE find(NODE a, TREE nodes[]);
        {
            TREE x;

(1)         x = nodes[a];
(2)         while (x->parent != NULL)
(3)             x = x->parent;
(4)         return x;
        }


        /* 让根节点较低的树成为根节点较高的树的子树，
           从而将根节点分别
           为x和y的树合并为一棵树 */
        void merge(TREE x, TREE y)
        {
            TREE higher, lower;

(5)         if (x->height > y->height) {
(6)             higher = x;
(7)             lower = y;
            }
            else {
(8)             higher = y;
(9)             lower = x;
            }
(10)        lower->parent = higher;
(11)        if (lower->height == higher->height)
(12)            ++(higher->height);
        }
```

图9-19　辅助函数find和merge

第二个函数是merge，[1]它接受指向两个树节点的指针x和y，要让函数正常工作，它们一定是需要合并的两棵树的根节点。第(5)行的测试确定了哪个根节点的高度更大，如果相等就直接

① 不要把该函数与第2章和第3章中用于归并排序的同名函数弄混了。

选择y。在第(6)行至第(7)行或第(8)行至第(9)行，会根据具体的情况将较高的根节点赋值给局部变量higher，而较低的根节点则被赋值给局部变量lower。接着在第(10)行较低的根节点会成为较高根节点的子节点，而在第(11)行和第(12)行，如果T_1和T_2的高度相等，较高根节点（也就是现在合成的树的根节点）的高度要增加1。较低根节点的高度保持不变，不过现在这个值已经没有意义了，因为较低根节点现在已经不再是根节点了。

找出连通分支的算法的核心内容如图9-20所示。假设函数makeEdges()会把手头的图转换成由图中的边组成的链表，这里并未展示该函数的代码。

```
      #include <stdio.h>
      #include <stdlib.h>

      #define MAX 13
      typedef int NODE;
      typedef struct EDGE *EDGELIST;
      struct EDGE {
          NODE node1, node2;
          EDGELIST next;
      };

      typedef struct TREENODE *TREE;
      struct TREENODE {
          int height;
          TREE parent;
      };

      TREE find(NODE a, TREE nodes[]);
      void merge(TREE x, TREE y);
      EDGELIST makeEdges();

      main()
      {
          NODE u;
          TREE a, b;
          EDGELIST e;
          TREE nodes[MAX];

          /* 初始化节点，使得每个节点都在由其自身构成的树中 */
(1)       for (u = 0; u < MAX; u++) {
(2)           nodes[u] = (TREE) malloc(sizeof(struct TREENODE));
(3)           nodes[u]->parent = NULL;
(4)           nodes[u]->height = 0;
          }

          /* 将e初始化为存放图中各边的表 */
(5)       e = makeEdges();

          /* 检查每条边，如果边的的端点在不同组分中，就将它们
             合并 */
(6)       while (e != NULL) {
(7)           a = find(e->node1, nodes);
(8)           b = find(e->node2, nodes);
(9)           if (a != b)
(10)              merge(a, b);
(11)          e = e->next;
          }
      }
```

图9-20　用来找出连通分支的C语言程序

用来找出连通分支的更优算法

我们会在9.6节中研究深度优先搜索时看到，其实还有更好的方法来计算连通分支，它只需要花费$O(m)$的时间，而不是$O(m \log n)$的时间。不过，9.4节中给出的数据结构本身也很实用，我们在9.5节中就要看到另一个使用该数据结构的程序。

图9-20的第(1)行到第(4)行会浏览数组`nodes`，而在第(2)行会为每个节点创建一个树节点。在第(3)行其`parent`字段会被置为`NULL`，表示它是其自身构成的树的根节点，而在第(4)行会将其`height`字段置为0，以反映出在由该节点自己构成的树中就只有它这一个节点。

然后第(5)行会把e初始化为指向边链表中的第一条边，而且第(6)行到第(11)行的循环会一次检查每一条边。在第(7)行和第(8)行，我们找到了当前边两个端点的根节点。接着在第(9)行要测试这些根节点是否为不同的树节点。如果是，那么当前边的两个端点在不同分支中，而且我们要在第(10)行合并这些分支。如果该边的两个端点在同一分支中，就跳过第(10)行，因此就不会对树集合造成任何改变。最后，第(11)行会带着我们沿着边链表行进。

9.4.4 连通分支算法的运行时间

我们来确定一下图9-20所示的算法处理一幅图要花多长时间。假设该图有n个节点，并设节点数和边数的较大者为m。[①]首先看看这些辅助函数。我们论证过，将高度较低的树合并到高度较高的树中的策略可以保证从任意树节点到达其根节点的路径都不会比$\log n$长。因此，`find`会花费$O(\log n)$的时间。

接下来要查看图9-19中的函数`merge`。它的每条语句都花费$O(1)$的时间。因为其中不含循环或函数调用，所以整个函数也只花$O(1)$的时间。

最后来看看图9-20所示的主程序。第(1)行到第(4)行的`for`循环循环体要花费$O(1)$的时间，而且该循环要迭代n次。因此，第(1)行到第(4)行所花的时间是$O(n)$。假设第(5)行要花费$O(m)$的时间。最后，考虑一下第(6)行到第(11)行的`while`循环。在循环体中，第(7)和第(8)行每行都要花费$O(\log n)$的时间，因为它们都调用了函数`find`，而我们刚刚已经确定过`find`要花费$O(\log n)$的时间。第(9)行和第(11)行显然只要$O(1)$的时间。第(10)行同样只要$O(1)$的时间，因为我们刚刚确定了`merge`花费$O(1)$的时间。因此，整个循环体要花费$O(\log n)$的时间。而`while`循环会迭代m次，其中m是边的数量。因此，该循环的运行时间就是$O(m \log n)$，也就是迭代的次数乘以循环体运行时间的边界。

然后，一般来说，整个程序的运行时间可以表示为$O(n+m+m \log n)$。不过，m至少是n，所以$m \log n$这一项就主导了其他两项。因此，图9-20所示程序的运行时间就是$O(m \log n)$。

9.4.5 习题

(1) 图9-21列出了密歇根州的一些城市以及它们之间的公路里程数。就本习题的目的而言可以忽略里程数。以本节描述的方式检查每条边，构建该图的连通分支。

(2) * 通过对k的归纳证明，有k个节点的连通分支至少有$k-1$条边。

① 把m当作边的数量是很正常的，不过在某些图中，节点比边更多。

城市1	城市2	距 离
马凯特（Marquette）	苏圣玛丽（Sault Ste. Marie）	153
萨吉诺（Saginaw）	弗林特（Flint）	31
大急流城（Grand Rapids）	兰辛（Lansing）	60
底特律（Detroit）	兰辛（Lansing）	78
埃斯卡诺巴（Escanaba）	苏圣玛丽（Sault Ste. Marie）	175
安娜堡（Ann Arbor）	底特律（Detroit）	28
安娜堡（Ann Arbor）	巴特尔克里克（Battle Creek）	89
巴特尔克里克（Battle Creek）	卡拉马祖（Kalamazoo）	21
梅诺米尼（Menominee）	埃斯卡诺巴（Escanaba）	56
卡拉马祖（Kalamazoo）	大急流城（Grand Rapids）	45
埃斯卡诺巴（Escanaba）	马凯特（Marquette）	78
巴特尔克里克（Battle Creek）	兰辛（Lansing）	40
弗林特（Flint）	底特律（Detroit）	58

图9-21 密歇根州某些城市间的距离

(3) * 有一种更简单的方法实现“合并”和“寻找”，在使用这种方法时，要使用以节点为索引的数组，给出每个节点的分支。一开始，每个节点都在由它自己构成的分支中，而且我们要用相应的节点来为这种分支命名。要找到节点的分支，只要查找对应的数组项即可。要合并分支，就要沿数组向下行进，将所有出现第一个分支的地方都改为第二个分支。
   (a) 编写C语言程序实现这一算法。
   (b) 该程序的运行时间是多少？将其表示为节点数n与节点数和边数较大值m的函数。
   (c) 对某些边数和节点数而言，这种实现其实比本章中描述的实现还要好。什么时候这种实现更好？

(4) * 假设本节的连通分支算法中不是将较低的树合并到较高的树中，而是将节点较少的树合并到节点较多的树中。这种连通分支算法的运行时间是否仍为$O(m \log n)$？

9.5 最小生成树

连通分支问题有个很重要的推广，其中给定了以数字（整数或实数）作为边标号的无向图。我们不仅要找到连通分支，而且要为各分支找到连接分支中各节点的树。此外，该树一定是最小的，意味着边标号的和是尽可能小的。

这里讨论的树与第5章讨论过的树不太一样。这里的树中没有节点会被指定为根节点，而且没有子节点或子节点次序的概念。本节中提到“树”时，指的是没有根没有次序的树，就是那些不含简单环路的无向图。

无向图G的生成树是由G的节点与G的边的子集按照如下要求一起构成的。

(1) 连通节点，也就是说，任意两个节点之间都存在只用生成树中的边构成的路径。

(2) 形成无根且无次序的树，也就是说，树中没有（简单）环路。

如果G是单个连通分支，就总是存在生成树。最小生成树是给定图对应的任意生成树中边标号的和最小的那个。

✦ 示例 9.16

设图G是如图9-4或图9-10所示对应瓦胡岛的连通分支。图9-22展示了一种可能的生成树。它是通过删除{迈里，瓦西阿瓦}和{卡内奥赫，拉耶}这两条边并剩下其余5条边形成的。这棵树的权（也就是边标号之和）为84。正如我们将要看到的，这不是最小值。

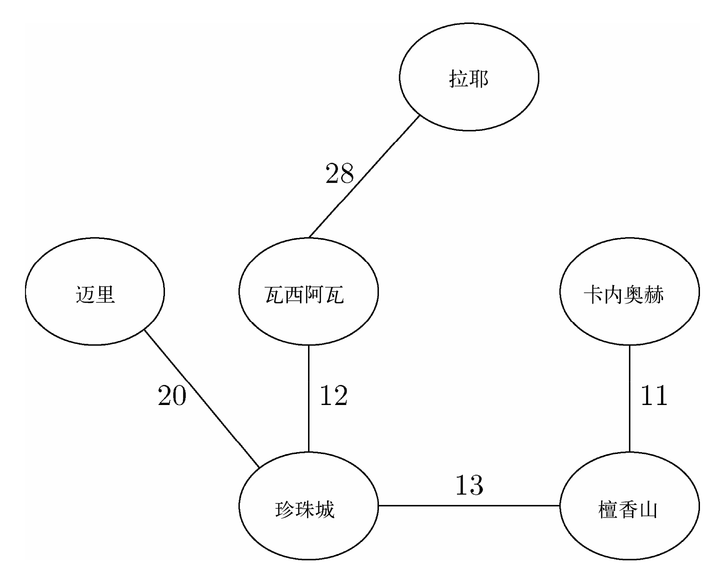
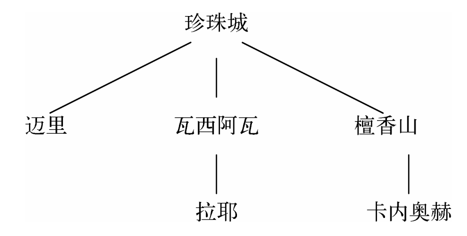


图9-22 对应瓦胡岛的生成树

有根树与无根树

无根树的概念似乎不应该很奇怪。其实我们可以从无根树中任选一个节点作为根节点。这样就为所有的边给出了远离根节点，或者是从父节点到子节点的方向。从物理意义上讲，这就像是从无根树的某个节点提起该树，让该树其他部分从选定的节点处吊起来。例如，可以将珍珠城作为图9-22所示生成树的根节点，它就成了下面这样



如果愿意的话，可以为每个节点的子节点排定次序，不过这种次序是任意的，与原来的无根树之间没有关系。

9.5.1 找到最小生成树

有多种用于找到最小生成树的算法。我们将研究其中一种，名为克鲁斯卡尔算法（Kruskal's

algorithm)，它是对9.4节中讨论过的寻找连通分支算法的一种简单扩展。需要进行的修改如下所述。

(1) 需要按照边标号的递增次序考虑这些边。我们在示例9.14中刚好选择了这种次序，不过对连通分支而言这不是必要的。

(2) 在考虑边时，如果边的两个端点在不同分支中，就要选择该边构成生成树并且合并两个分支，正如9.4节的算法中所做的。否则，就不选择这条边构成生成树，当然也就不用合并分支。

✦ 示例 9.17

Acme Surfboard Wax 公司在如图9-4所示的13个城市中都有办公地点。它希望从电话公司租用专用的数据传输线路，我们假设电话线路是沿着图9-4中的边表示的公路架设的。在不同的岛屿之间，该公司必须使用卫星传输，而成本与分支数量是成正比的。不过，对地面传输线路来说，电话公司是按里程收费的。[①]因此，我们希望为图9-4所示的图中各连通分支找出最小生成树。

如果按照分支分开这些边，就可以分别为各分支运行克鲁斯卡尔算法。不过，如果我们尚不知道有哪些分支，就必须将所有的边放在一起考虑，从最小的标号开始，按照图9-14的次序进行。正如9.4节中那样，我们先从由节点本身构成的分支中的各节点开始。

首先考虑标号最小边{卡内奥赫，檀香山}。这条边将这两个城市合并到一个分支中，而且因为我们执行了合并操作，所以就选择了该边用来构成最小生成树。2号边是{瓦西阿瓦，珍珠城}，而且因为这条边也是合并了两个分支，所以它也被选来构成该生成树。同样，第三条边{珍珠城，檀香山}和第四条边{瓦西阿瓦，迈里}也合并了分支，因此也被放入生成树。

第五条边{卡胡卢伊，凯奥凯阿}合并了这两个城市，而且也被接纳到生成树中，虽然这条边是要成为表示毛伊岛分支的生成树的一部分，而不是和前四条边那样是瓦胡岛分支的一部分。

第六条边{迈里，珍珠城}连接着已经出现在同一分支中的两个城市。因此，该边会被生成树拒之门外。即便我们必须选择某条标号更大的边，也不能选择{迈里，珍珠城}，因为这样一来就会在迈里、瓦西阿瓦和珍珠城间形成一条环路。在生成树中是不可以有环路的，所以这3条边中必须有一条被排除在外。随着我们按照标号的次序考虑这些边，最后的边肯定有着最大的标号，也是最佳方案要排除掉的。

第七条边{拉海纳，卡胡卢伊}和第八条边{拉耶，卡内奥赫}都被生成树接纳，因为它们合并了分支。而9号边{拉耶，瓦西阿瓦}会因为它的端点在同一分支中而不被接受。我们会接受10号边和11号边，它们形成了表示“大岛”分支的生成树，而且我们会接纳12号边以完成毛伊岛分支。13号边不会被接纳，因为它连接的科纳和希洛已经被10号边和11号边连接到同一分支中了。得到各分支的生成树如图9-23所示。

① 这是一种为租用的电话线路收费的可行方式。人们可以找出连接这些所需场所的最小生成树，且收费是根据该树的权得出的，而不用考虑提供电话连接的实际方式。

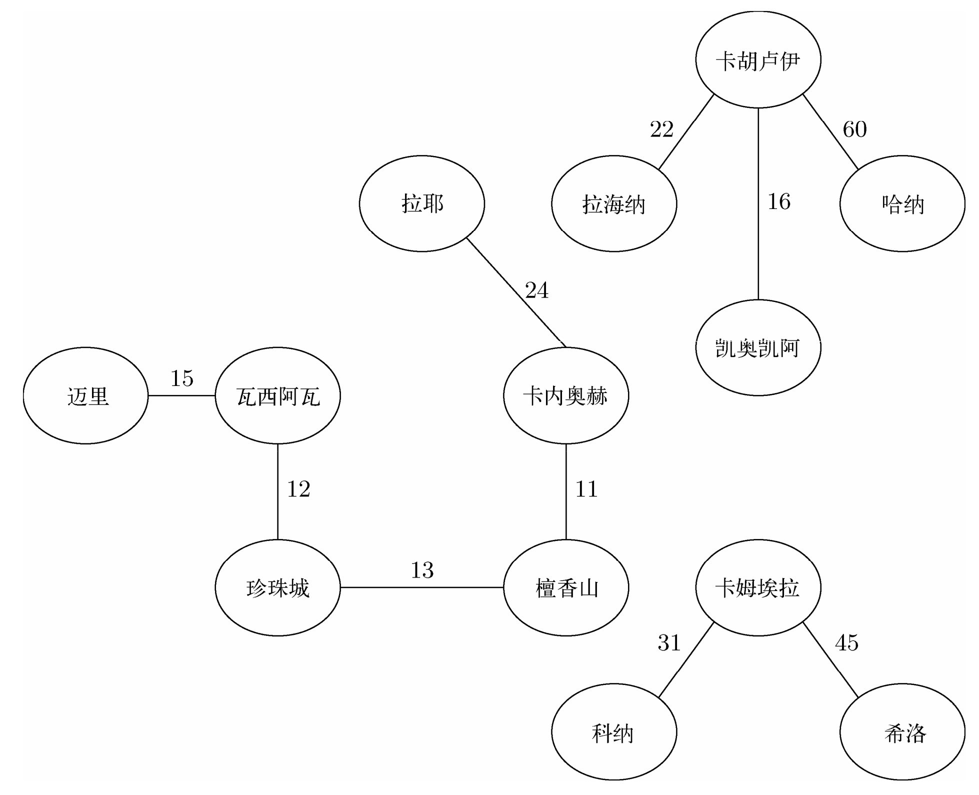


图9-23 图9-4所示的图对应的生成树

9.5.2 克鲁斯卡尔算法起效的原因

可以证明，克鲁斯卡尔算法可为某给定图生成权最小的生成树。设G是无向连通图。简便起见，如果需要，我们会为某些标号加上一些极小的量，使得所有标号都是不同的，而且添加的各极小量的和要小于G中任意不同标号之间的差。这样一来，带有新标号的G就会有唯一的最小生成树，它将会是带原有权的G所有最小生成树中的一棵。

接着，设e_1、e_2、…、e_m是G的所有边，而且是按照标号从小到大的顺序排列的。请注意，这个次序也是克鲁斯卡尔算法处理这些边依照的次序。设K是带有用克鲁斯卡尔算法生成的调整后标号的图G对应的生成树，并设T是G唯一的最小生成树。

我们要证明K和T其实是相同的。如果它们是不同的，一定至少存在一条边在其中一棵树而不在另一棵中。设e_i是这一系列边中第一条这样的边，也就是说，e_1、…、e_{i-1}要么同在K和T中，要么都不在K和T中。这里有两种情况，取决于e_i是在K中还是在T中。我们在每种情况下都能得出矛盾，因此就能得出e_i是不存在的，因此$K = T$，而且K是G的最小生成树。

贪婪有时是有用的

克鲁斯卡尔算法是贪婪算法的一个好例子，在贪婪算法中我们会做出一系列的决定，每次都做出当时最佳的选择。这些局部的决定是决定哪条边要被添加到正在成形的生成树中。在各情况下，我们都要选择那条标号最小但又不会因为产生环路而破坏“生成树”定义的边。通常，

局部最优选择的整体效果不是全局上最适合的。然而，在克鲁斯卡尔算法的情况中，可以证明结果从全局上讲也是最佳的，也就是一棵权最小的生成树。

情况1。边e_i在T中而不在K中。如果克鲁斯卡尔算法不接受e_i，那么e_i肯定与之前为K选择的边中某路径P形成了环路，如图9-24所示。因此，组成P的边都能在e_1、…、e_{i-1}中找到。不过，T和K对这些边是一致的，也就是说，如果P的边在K中，那么这些边也在T中。不过因为T中含有e_i，P加上e_i就在T中形成了环路，这与我们说T是生成树的假设是矛盾的。因此，e_i在T中而不在K中是不可能的。

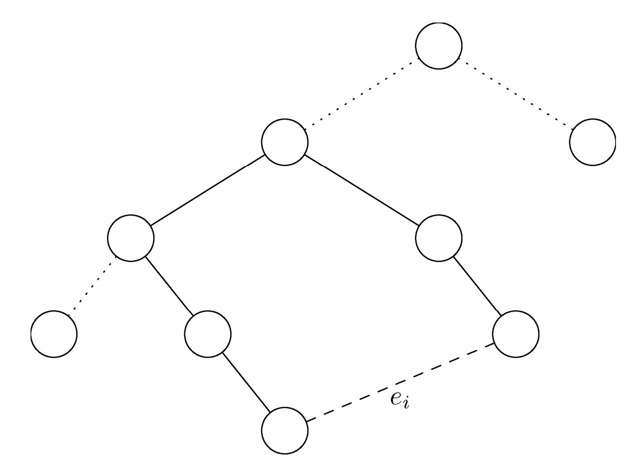


图9-24　路径P（实线）在T和K中，边e_i只在T中

情况2。边e_i在K中而不在T中。设e_i连接了节点u和v。因为T是连通的，所以在T中节点u和v之间一定存在某条无环路径，假设称其为Q。因为Q没有用到边e_i，所以Q加上e_i在图G中形成了简单环路。这里存在两种子情况，具体取决于e_i的标号是否比路径Q上所有边的标号都大。

(a) 边e_i有着最高的标号。那么Q上的所有边都在$\{e_1，…，e_{i-1}\}$中。请记住，T和K在e_i之前的所有边都是一样的，所以Q中所有的边也是K中的边。不过e_i也在K中，这表示K是一条环路。因此我们排除了e_i的标号比Q中任何边的标号都高的可能。

(b) 路径Q上的某边f的标号比e_i的标号高。假设f连接节点w和x。图9-25展示了树T中的这种情况。如果将边f从T中删除，并加上边e_i，就不会形成环路，因为路径Q因f被删除而中断了。得到的边的集合权要比T低，因为f有着比e_i更高的标号。我们声明得到的这些边仍然连通所有节点。要知道原因，请注意w和x仍然是连通的，有一条路径沿着Q从w到u，然后沿着边e_i，然后再沿着路径Q从v到x。因为$\{w，x\}$是唯一一条被删除的边，如果它的终点仍然是连通的，那么显然所有节点都是连通的。因此，边的新集合是生成树，而它的存在与T是最小生成树的假设相矛盾。

现在就已经证明了e_i不可能在K中而不在T中。这样就排除了第二种情况。因为e_i不可能在T和K中，所以可以得出结论，K其实就是最小生成树T。也就是说，克鲁斯卡尔算法总是能找到最小生成树。

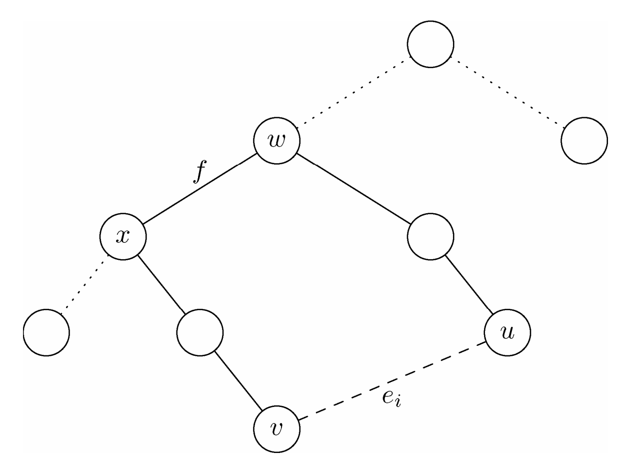


图9-25　路径Q（实线）在T中，我们可以将边e_i添加到T中并删除边f

9.5.3　克鲁斯卡尔算法的运行时间

假设对某一含n个节点的图运行克鲁斯卡尔算法。就像9.4节那样，设m是节点数与边数的较大者，但请记住，通常边数是较大者。假设该图是用邻接表表示的，这样就可以在$O(m)$的时间内找到所有的边。

首先，必须用标号为边排序，如果使用了诸如归并排序这样的高效排序算法，就要花上$O(m \log m)$的时间。接着要考虑这些边，花上$O(m \log n)$的时间进行所有的合并与寻找，就像在9.4节中讨论过的那样。因此看起来克鲁斯卡尔算法的总运行时间是$O(m(\log n+\log m))$。

不过，要注意到$m \leqslant n^2$，因为只存在$n(n-1)/2$个节点对。因此，$\log m \leqslant 2\log n$，这样一来$m(\log n+\log m) \leqslant 3m\log n$。因为在大$O$表达式中常数因子是可以省略掉的，所以可以得出结论：克鲁斯卡尔算法的运行时间是$O(m \log n)$。

9.5.4　习题

(1) 如果瓦西阿瓦被选为根节点，画出表示图9-22的树。

(2) 使用克鲁斯卡尔算法为边和标号都如图9-21（见9.4节习题）所示的各分支找到最小生成树。

(3) ** 证明，如果图G是有n个节点的无向连通图，而且T是G的生成树，则T有$n-1$条边。提示：我们需要对n进行归纳。难点在于证明T一定有某个度为1的节点v，也就是说，T刚好只有一条边含节点v。考虑如果对每个节点u都至少有两条T的边含有u会发生什么。沿着边进出一系列的节点，最终会找到一条环路。因为假设T是生成树，所以它不可能含有环路，这样一来就形成矛盾了。

(4) * 一旦我们选定了$n-1$条边，就不需要考虑将更多的边纳入该生成树了。描述克鲁斯卡尔算法的一个变种，它不会为所有边排序，但会将它们放入优先级队列中，将边标号的相反数作为其优先级（也就是最短的边会首先被*deleteMax*选中）。证明，如果生成树可以在前$m/\log m$条边中找到，那么这一版本的克鲁斯卡尔算法就只需要花$O(m)$的时间。

(5) * 假设为图G找到了最小生成树T，然后向G添加权为w的边$\{u, v\}$。在什么情况下T仍是新图的最小生成树？

(6) ** 无向图G的*欧拉回路*是起止点为同一节点而且刚好含有图G中每条边一次的路径。

(a) 证明，当且仅当每个节点都为偶数度时，无向连通图含有欧拉回路。

(b) 设G是有m条边而且每个节点都为偶数度的无向图。给出运行时间是$O(m)$的为图G构建欧拉回路的算法。

9.6 深度优先搜索

我们现在要描述一种对有向图而言很实用的图探索方法。在5.4节中我们讨论过树的前序遍历和后序遍历，其中从根节点开始，递归地探索了访问过的每个节点的子节点。我们几乎可以将同样的思路应用到任意有向图上。[①]从任意节点出发，可以递归地探索其后继。

不过，必须小心图中存在环路的情况。如果存在环路，我们可能会绕着环路永远地递归调用探索函数。例如，考虑图9-26中的图。从节点a开始，我们可能决定接下来探索节点b。从b出发可能会先探索c，然后从c出发可能要先探索b。这样就会导致无限递归，反复地探索b和c。其实，我们选择按照什么次序探索b和c的后继是不重要的。要么会困在其他的环路中，要么最终会无限地从b探索c并从c探索b。

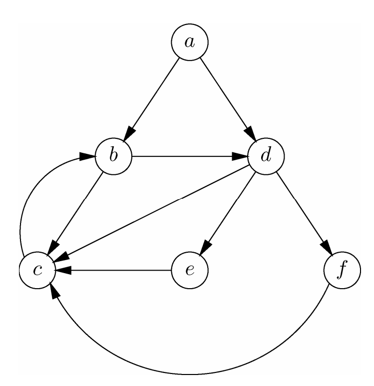


图9-26　有向图的示例

这一问题有个简单的解决方案：在访问节点的过程中为其做上标记，并永不再次访问标记过的节点。这样一来，我们从起始节点起可以到达的任何节点都会被探索到，而之前已经访问的节点不会被再次访问。我们将看到这种探索所花的时间是与被探索的弧的数量成比例的。

这种搜索算法叫作深度优先搜索，因为我们会尽可能快地行进到离初始节点尽可能远（尽可能"深"）的节点。这可以通过一种简单的数据结构实现。这里要再次假设使用NODE类型为节点命名，而且该类型就是int类型。我们用邻接表表示弧。因为需要为每个节点添加一个"标记"，其值是从VISITED和UNVISITED中二选一，所以要创建一个结构体数组来表示该图。这些结构体要同时包括这里所说的标记以及邻接表的表头。

```
enum MARKTYPE {VISITED, UNVISITED};
typedef struct {
    enum MARKTYPE mark;
    LIST successors;
} GRAPH[MAX];
```

其中LIST为邻接表，是按照以下习惯方式定义的

```
typedef struct CELL *LIST;
struct CELL {
    NODE nodeName;
    LIST next;
};
```

① 请注意，如果将树中的弧看作存在从父节点到子节点的方向，树就可以被当作有向图的一个特例。其实，树还总是无环图。

一开始要将所有的节点标记为UNVISITED。图9-27所示的递归函数dfs(u,G)会处理某幅在外部定义的GRAPH类型的图G中的节点u。

在第(1)行我们将u标记为VISITED，这样就不用再次对它调用dfs了。第(2)行会初始化指针p，它指向节点u的邻接表的第一个单元。第(3)行至第(7)行的循环会带p沿着邻接表向下行进，依次考虑u的各后继v。

```
     void dfs(NODE u, GRAPH G)
     {
         LIST p; /* 沿着u对应的邻接表下行 */
         NODE v; /* 由p指向的单元中存放的节点 */

(1)      G[u].mark = VISITED;
(2)      p = G[u].successors;
(3)      while (p != NULL) {
(4)          v = p->nodeName;
(5)          if (G[v].mark == UNVISITED)
(6)              dfs(v, G);
(7)          p = p->next;
         }
     }
```

图9-27　递归的深度优先搜索函数

第(4)行会将v置为节点u“当前”的后继。在第(5)行，我们会测试v之前是否已经被访问过。如果是，就跳过第(6)行的递归调用并在第(7)行中将p移动到邻接表的下一个单元。不过，如果v从未被访问过，就要在第(6)行从节点v开始进行深度优先搜索。最后，完成对dfs(v,G)的调用。然后执行第(7)行，让p沿着u的邻接表向下移动并进行循环。

✦　示例 9.18

假设G是图9-26所示的图，而且为了简化问题，假设各邻接表中的节点都是按照字母表顺序排列的。一开始，所有节点都会被标记上UNVISITED。调用dfs(a)，[①]节点a在第(1)行会被标记为VISITED，而且我们在第(2)行要初始化指针p，它指向a的邻接表的第一个单元。在第(4)行v被置为b，因为b是第一个单元中的节点。由于b当前处于未被访问的状态，所以第(5)行的测试会成功，并且要在第(6)行调用dfs(b)。

现在，要以b为参数开始一次对dfs的新调用，而$u=a$的旧调用处于休眠状态而并未终止。因为c是b的邻接表中的第一个节点，所以在第(4)行c成了v的值。节点c是未访问过的，所以我们在第(5)行会成功并在第(6)行调用dfs(c)。

现在激活了对dfs的第三次调用，而且要开始dfs(c)，我们将c标记为VISITED，并在第(4)行将v置为b，因为b是c的邻接表中第一个也是唯一的一个节点。不过，b已经在对dfs(b)的调用的第(1)行中被标记为VISITED了，所以我们要跳过第(6)行，并在第(7)行将p沿着c的邻接表向下移动。因为c没有更多后继了，这样p就成了NULL，所以第(3)行的测试就会失败，对dfs(c)的调用就完成了。

现在又回到对dfs(b)的调用。指针p在第(7)行被前移，现在它指向b的邻接表的第二个单元，这个单元存放着节点d。我们在第(4)行将v置为d，因为d是未被访问过的，所以在第(6)行要调用dfs(d)。

在执行dfs(d)时，我们会将d标记为VISITED。那么v首先会被置为c。但因为c是被访问过的，

① 在接下来的内容中，我们将省略dfs的第二个参数，因为它永远都是图G。

因此下一次进行循环时会有$v=e$。这会引发对`dfs(e)`的调用。节点e只有c这么一个后继，所以在将e标记为`VISITED`后，`dfs(e)`就会返回到`dfs(d)`。接下来在`dfs(d)`的第(4)行进行$v=f$的赋值，并调用`dfs(f)`。在把f标记为`VISITED`后，我们会发现f也只有c这么一个后继，而c又是被访问过的。

现在就完成了对`dfs(f)`的调用。因为f是d的最后一个后继，所以我们完成了对`dfs(d)`的调用，而且由于d是b的最后一个后继，这样也就完成了对`dfs(b)`的调用。这样就把我们带回了`dfs(a)`。节点a还有另一个后继d，不过该节点是被访问过的，所以我们也就完成了对`dfs(a)`的调用。

图9-28总结了`dfs`对图9-26所示图的操作。我们展示了对`dfs`进行调用的情况，并在右侧给出了当前处于活跃状态的调用。我们还表示了每一步执行的活动，并展示了与当前活跃的调用相关联的局部变量v的值，或者是给出`p = NULL`，表示没有相应的v值。

dfs(a) $v=b$				调用 dfs(b)
dfs(a) $v=b$	dfs(b) $v=c$			调用dfs(c)
dfs(a) $v=b$	dfs(b) $v=c$	dfs(c) $v=b$		跳过，b已经被访问过
dfs(a) $v=b$	dfs(b) $v=c$	dfs(c) $p=$NULL		返回
dfs(a) $v=b$	dfs(b) $v=d$			调用dfs(d)
dfs(a) $v=b$	dfs(b) $v=d$	dfs(d) $v=c$		跳过，c已经被访问过
dfs(a) $v=b$	dfs(b) $v=d$	dfs(d) $v=e$		调用dfs(e)
dfs(a) $v=b$	dfs(b) $v=d$	dfs(d) $v=e$	dfs(e) $v=c$	跳过，c已经被访问过
dfs(a) $v=b$	dfs(b) $v=d$	dfs(d) $v=e$	dfs(e) $p=$NULL	返回
dfs(a) $v=b$	dfs(b) $v=d$	dfs(d) $v=f$		调用dfs(f)
dfs(a) $v=b$	dfs(b) $v=d$	dfs(d) $v=f$	dfs(f) $v=c$	跳过，c已经被访问过
dfs(a) $v=b$	dfs(b) $v=d$	dfs(d) $v=f$	dfs(f) $p=$NULL	返回
dfs(a) $v=b$	dfs(b) $v=d$	dfs(d) $p=$NULL		返回
dfs(a) $v=b$	dfs(b) $p=$NULL			返回
dfs(a) $v=d$				跳过，d已经被访问过
dfs(a) $p=$NULL				返回

图9-28　在深度优先搜索期间所执行调用的记录

9.6.1 构建深度优先搜索树

因为我们标记了节点以防访问它们两次，这样一来在探索图的过程中图就像树那样了。其实，也可以绘出一棵树，其父子节点之间的边就是被搜索的图G中的某些弧。如果我们在对`dfs(u)`的调用中，而且它会带来对`dfs(v)`的调用，那么我们就让v成为u在该树中的子节点。u的子节点是按照对这些子节点调用*dfs*的次序从左向右出现的。而第一次*dfs*调用所针对的节点就是该树的根节点。不会对任何节点调用*dfs*两次，因为在第一次调用后这些节点就会被标记为VISITED。因此，这样定义的结构真是棵树。我们可以称这样的树是某给定图的深度优先搜索树（depth-first search tree）。

✦ 示例 9.19

图9-29所示的树展示了图9-28总结的对图9-26所示的图的探索过程。我们把代表父子关系的树向弧（tree arc）表示为实线，图中的其他弧被表示为虚线箭头。这里我们应该忽略节点标号的数字。

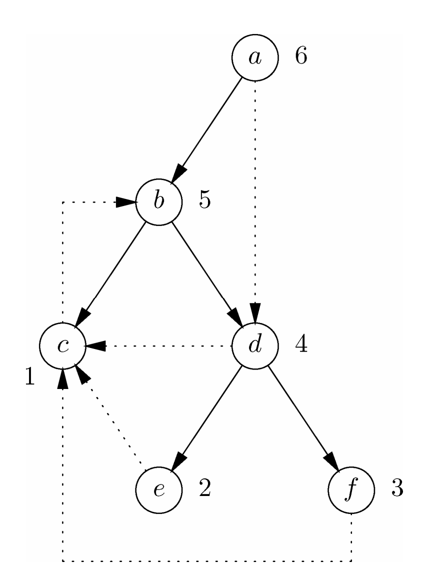


图9-29 图9-26所示的图的一种可能的深度优先搜索树

9.6.2 深度优先搜索树弧的分类

当我们为图G构建深度优先搜索树时，可以把G中的弧分为4组。应该不难理解，这种分类是就某棵特定的深度搜索树而言的，或者说，是针对各邻接表中节点的某种特定次序（形成对G的一次特定探索）而言的。这4类弧分别如下。

(1) 树向弧，满足`dfs(v)`被`dfs(u)`调用的弧$u \to v$。

(2) 前向弧（forward arc），满足v是u的真子孙但又不是u的子节点的弧$u \to v$。例如，在图9-29中，弧$a \to b$就是唯一的前向弧。树向弧都不是前向弧。

(3) 后向弧（backward arc），满足v是u在该树中的祖先（$u = v$也是可以的）的弧$u \to v$。图9-29中，弧$c \to b$是唯一的后向弧。任何自环，也就是节点到其自身的弧都被分类为后向弧。

(4) 横向弧（cross arc），满足v既不是u的祖先也不是其子孙的弧$u \to v$。图9-29中有3条这样的弧：$d \to c$、$e \to c$和$f \to c$。

在图9-29中，每条横向弧都是从右至左的。这种情况并非巧合。假设在某深度优先搜索树中有一条横向弧$u \to v$满足u在v的左侧。考虑一下在调用`dfs(u)`期间会发生什么。到完成对`dfs(u)`的调用之时，我们应该已经考虑过从u到v的弧了。如果v尚未被放置到树中，那么它就会成为u在该树中的子节点。因为这种情况显然不会发生（这样v就不会在u的右侧了），所以在考虑弧$u \to v$时，v肯定已经在该树中了。

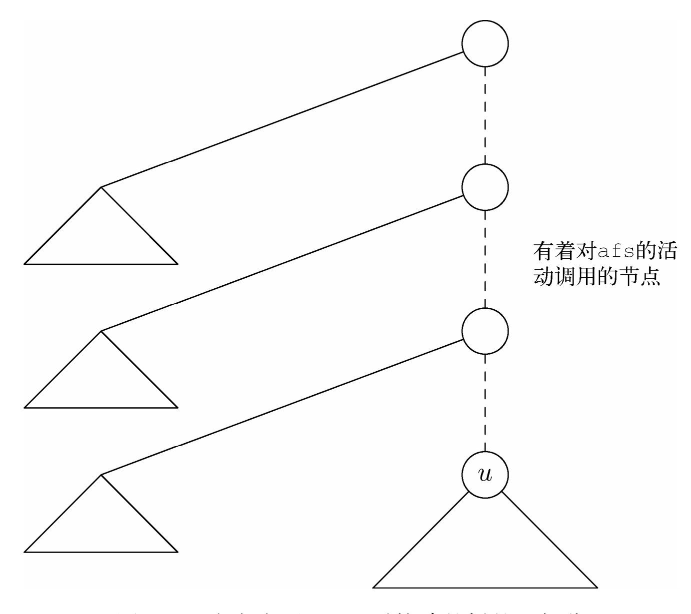


图9-30　在考虑弧$u \to v$时构建的树的一部分

不过，图9-30展示了当`dfs(u)`处于活动状态时存在的树的一部分。因为子节点会按照从左至右的次序添加进来，所以迄今为止节点u的真祖先没有子节点在u的右侧。因此，v只可能是u的祖先，u的子孙，或者是在u左侧的某个位置。因此，如果$u \to v$是横向弧，v就一定是在u的左侧，而不可能像我们最初假设的那样在u的右侧。

9.6.3　深度优先搜索森林

在示例9.19中，我们特别幸运，从节点a开始，就能够到达图9-26所示图的全部节点。但假设我们从其他节点开始，就可能没法到达a，a就不会出现在深度优先树中。因此，探索图的一般方式是构建一系列的树。我们从某个节点u开始并调用`dfs(u)`。如果还有节点未被访问过，就再选择一个节点，比方说是v，并调用`dfs(v)`。只要还有节点未被分配到任一树中，就继续重复该过程。

在所有节点都被分配到一棵树中后，我们就按照构建这些树的先后次序，把构建出的树从左到右列出来。这一列树就叫作*深度优先搜索森林*（depth-first search forest）。利用之前定义的`NODE`和`GRAPH`数据类型，可以通过图9-31所示的函数，从所需的那么多根节点开始搜索，对完全从外部定义的图G进行探索。这里我们假设`NODE`类型为`int`类型，而且MAX是G中的节点数。

```
     void dfsForest(GRAPH G);
     {
         NODE u;

(1)      for (u = 0; u < MAX; u++)
(2)          G[u].mark = UNVISITED;
(3)      for (u = 0; u < MAX; u++)
(4)          if (G[u].mark == UNVISITED)
(5)              dfs(u, G);
     }
```

图9-31 通过探索所需的那么多树对图进行探索

在第(1)行和第(2)行中，我们会把所有节点初始化为UNVISITED。然后，在第(3)行到第(5)行的循环中，要依次考虑各节点u。在考虑u时，如果该节点尚未被添加到任何树中，那么在进行第(4)行的测试时它仍然会被标记为未被访问过。在这样的情况下，我们就会在第(5)行调用`dfs(u,G)`，并探索以u为根节点的深度优先搜索树。特别要说的是，第一个节点总是会成为树的根节点。不过，如果在执行第(4)行的测试时u已经被添加到树中，那么u就会被标记为`VISITED`，因此不会创建以u为根节点的树。

✦ 示例 9.20

假设将上述算法应用到图9-26所示的图上，但是设d是名称为0的节点，也就是说，d是该深度搜索生成森林中树的第一个根节点。调用`dfs(d)`，这会构建图9-32中的第一棵树。现在除了a之外的所有节点都已经被访问过。当u在图9-31第(3)行到第(5)行的循环中成为各节点时，除了$u = a$时之外第(4)行的测试都会失败。然后，我们会创建如图9-32所示的单节点第二棵树。请注意，在调用`dfs(a)`时，a的两个后继都带有`VISITED`标记，因此我们不再从`dfs(a)`进行任何递归调用。

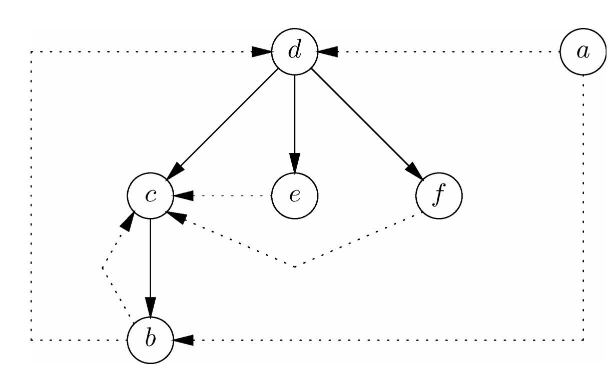


图9-32 深度优先搜索森林

当把图中的节点表示为深度优先搜索森林时，前向弧、后向弧与树向弧的概念还像之前那样。不过，横向弧的概念必须扩展到包含那些从一棵树到其左侧的树的弧。这种横向弧的例子包括图9-32中的 $a \to b$ 和 $a \to d$ 。

横向弧总是从右至左的这一规则继续成立。原因还是一样的。如果存在从一棵树到其右侧树的横向弧 $u \to v$ ，那么考虑一下当调用`dfs(u)`时会发生什么。因为v没有被添加到当时正在形成的树中，所以它肯定已经在某棵树中。但是u右侧的树尚未创建，因此v不可能是这些树的一部分。

尽善尽美的深度优先搜索

无论节点数和弧数之间存在什么的关系，对图的深度优先探索需要的时间是与图的“大小”（也就是节点数与弧数之和）成正比的。因此，深度优先搜索与其他任意“查看”图的算法在速度上只有常数因子的差异。

9.6.4 深度优先搜索算法的运行时间

设G是有n个节点的图，并设m是节点数与弧数之间的较大者。图9-31所示的`dfsForest`就要花$O(m)$的时间。这一事实的证明需要一点小花招。在计算对`dfs(u)`的调用所花的时间时，我们不会将图9-27第(6)行中对`dfs`的递归调用所花的时间计算在内，就像3.9节中所建议的那样。不过，不难看出要为每个u值调用`dfu(u)`一次。因此，如果将每次调用的开销加起来，除掉其递归调用，就能得到将所有调用视作一个整体所花的总时间。

请注意，除掉在对`dfs`的递归调用中所花的时间，图9-27中第(3)行到第(7)行的`while`循环所花的时间是可以变化的，因为节点u的后继数量可能是从0到n的任一数字。假如设m_u是节点u的出度，也就是u的后继的数量。那么在执行`dfs(u)`期间进行该循环的次数就肯定是m_u。在评估`dfs(u)`的运行时间时，并不会把第(6)行执行`dfs(v,G)`的时间计算在内，而除该调用之外，整个循环体只要花$O(1)$的时间。因此，除去在递归调用上花的时间，第(3)行到第(7)行的循环花的总时间就是$O(1+m_u)$，这个附加的1是必要的，因为m_u可能为0，在这种情况下我们仍然需要为第(3)行的测试花上$O(1)$的时间。因为第(1)行和第(2)行的`dfs`要花$O(1)$的时间，所以可以得出结论，忽略递归调用，`dfs(u)`要花$O(1+m_u)$的时间完成调用。

现在可以看到，在运行`dfsForest`期间，刚好要为每个u值调用`dfs(u)`一次。因此，花在所有这些调用上的总时间是花在每次调用上的时间之和的大O，也就是$O(\sum_u(1+m_u))$。但是$\sum_u m_u$就是图中弧的数量，也就是最多为m，[①]因为每条弧都是都某一个节点发出的。节点数为n，所以$\sum_u 1$就是n。由于$n \leqslant m$，因此所有对`dfs`的调用花的总时间就是$O(m)$。

最后，必须考虑`dfsForest`花的时间。图9-31所示的该程序由各要迭代n次的两个循环组成。不难看出，除去对`dfs`的调用，循环体所花的时间是$O(1)$，因此这些循环的开销都是$O(n)$。这一时间会被对`dfs`的调用所花的$O(m)$时间主导。因为我们已经弄清了`dfs`调用所花的时间，所以可以得到`dfsForest`，再加上其所有对`dfs`的调用，要花$O(m)$的时间。

9.6.5 有向图的后序遍历

一旦有了深度优先搜索树，就可以按后序为其节点编号。不过，还有一种在搜索期间进行编号的简单方法。只要把为节点u加上编号当作`dfs(u)`完成前我们要做的最后一件事即可。然后，在节点的所有子节点被编号后，它自己就会被编上号，正好是按照后序编号的。

✦ 示例 9.21

图9-29所示的树，也就是我们对图9-26中的图进行深度优先搜索所建立的树，有着后序编号的节点标号。如果查看图9-28的过程，就会发现最先要返回的调用是`dfs(c)`，而且节点c会

① 其实，m_u的和刚好是m，除非节点数大于弧数。回想一下，m是节点数与弧数间的较大者。

被编为1号。然后我们会访问d，接着是e，并从对e的调用返回。因此，e的编号是2。同样，我们会访问f并从其返回，它会被编为3号。至此，已经完成了对d的调用，它会得到4这个编号。这样就完成了对dfs(b)的调用，因此b的编号就是5。最后，最开始对a的调用返回，给了a编号6。请注意，这一次序刚好就是我们以后序遍历该树会得到的。

我们可以对目前所编写的深度优先算法进行一些简单改动，从而为节点指定后序编号，这些改动如图9-33所总结。

```
        int k;/* 为已访问过的节点计数 */

        void dfs(NODE u, GRAPH G)
        {
            LIST p; /* 指向u的邻接表的单元 */
            NODE v; /* 由p指向的单元中存放的节点 */

 (1)        G[u].mark = VISITED;
 (2)        p = G[u].successors;
 (3)        while (p != NULL) {
 (4)            v = p->nodeName;
 (5)            if (G[v].mark == UNVISITED)
 (6)                dfs(v, G);
 (7)            p = p->next;
            }
 (8)        ++k;
 (9)        G[u].postorder = k;
        }

        void dfsForest(GRAPH G)
        {
            NODE u;

(10)        k = 0;
(11)        for (u = 0; u < MAX; u++)
(12)            G[u].mark = UNVISITED;
(13)        for (u = 0; u < MAX; u++)
(14)            if (G[u].mark == UNVISITED)
(15)                dfs(u, G);
        }
```

图9-33 以后序为有向图的节点编号的例程

(1) 在GRAPH类型中，需要为每个节点增加一个名为postorder的字段。对图G而言，我们要将节点u的后序编号放在G[u].postorder中。这一赋值是在图9-33的第(9)行完成的。

(2) 我们使用全局变量k按后序为节点计数。这一变量是在dfs和dfsForest的外部定义的。正如在图9-33中所见，我们在dfsForest的第(10)行将k初始化为0，并刚好在赋值后序编号之前，在dfs中的第(8)行将k递增1。

请注意，这样一来，当深度优先搜索森林中不止有一棵树时，第一棵树就会得到最低的编号，而紧接着的那棵树就会按顺序得到接下来的编号，以此类推。例如，在图9-32中，a会得到后序编号6。

9.6.6 后序编号的特殊属性

横向弧不能从左向右说明了与后序编号和图的深度优先表示中4种弧相关的一些有趣而实用的信息。在图9-34a中，我们看到图的深度优先表示中有u、v和w3个节点。节点v和w是u的子

孙，而且w在v的右侧。图9-34b展示了分别为这3个节点调用`dfs`各自的活动持续时间。

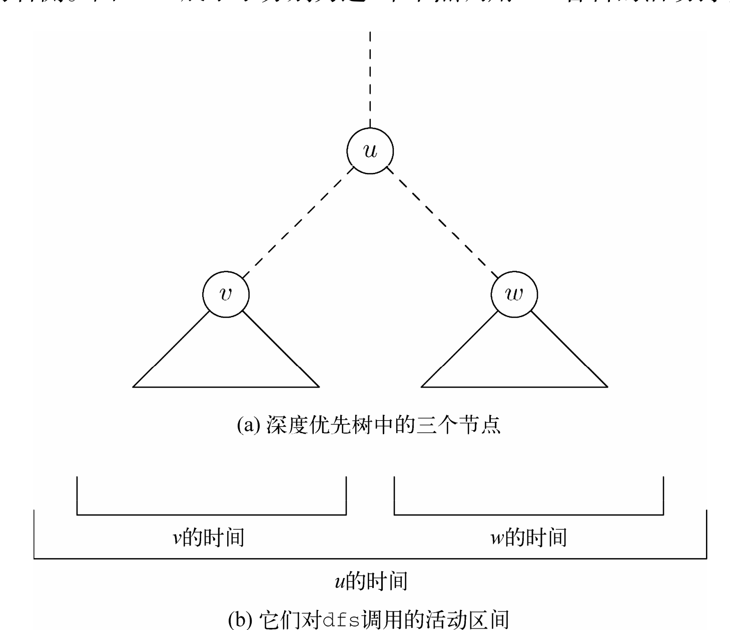


图9-34　树中位置间的关系和调用的持续时间

我们可以得出一些观点。首先，对子孙节点（比如v）进行的对`dfs`的调用，只在对祖先节点（比如u）的调用期间的某个子时间区间内是活动的。特别要指出的是，对`dfs(v)`的调用会在对`dfs(u)`的调用之前终止。因此，只要v是u的真子孙，v的后序编号肯定要比u的后序编号小。

其次，如果w在v的右侧，那么对`dfs(w)`的调用必须等到对`dfs(v)`的调用终止后才会开始。因此，只要v在w的左侧，v的后序编号就要比w的后序编号小。虽然图9-34中没有表示出来，但即便v和w在深度优先搜索森林的不同树中，只要v所在的树在w所在的树的左侧，同样的结论也是成立的。

我们现在可以为每条弧$u \to v$考虑u和v后序编号之间的关系了。

(1) 如果$u \to v$是树向弧或前向弧，那么v是u的子孙，所以v按后序要先于u。

(2) 如果$u \to v$是横向弧，那么我们知道v在u的左侧，因此按后序v还是先于u。

(3) 如果$u \to v$是后向弧而且$u \neq v$，那么v是u的真祖先，因此按后序v在u之后。不过，对后向弧而言$v = u$是有可能的，因为自环也是后向弧。因此，一般来说，对后向弧$u \to v$而言，我们知道v的后序编号是不会小于u的后序编号的。

总之，我们看到，弧头部按后序是要先于尾部的，除非该弧是后向弧，在弧是后向弧的情况中，尾部按后序是不会在头部之后的。因此，只要找到那些尾部按后序不大于头部的弧，就可以认定它们是后向弧。我们在9.7节中将看到这一概念的若干应用。

9.6.7　习题

(1) 为图9-5中的树（见9.2节的习题）给出两棵从节点a出发的深度优先搜索树。给出从节点d出现的深度优先搜索树。

(2) * 不管从图9-5中的哪个节点开始，我们最后都只得到深度优先搜索森林中的一棵树。简要解释对这幅特定的图为什么一定是这种情况。

(3) 对9.6节习题(1)中的各棵树，指出哪些弧是树向弧、前向弧、后向弧和横向弧。

(4) 对9.6节习题(1)中的各棵树，给出节点的后序编号。

(5) * 考虑含a、b、c这3个节点以及$a \to b$和$b \to c$这两条弧的图。为这幅图给出所有可能的深度优先搜索森林，为每棵树考虑所有可能的起始节点。每个森林的节点后序编号各是怎样的？

(6) * 考虑把习题(5)的图一般化为具有a_1、a_2、…、a_n这n个节点和$a_1 \to a_2$、$a_2 \to a_3$、…、$a_{n-1} \to a_n$这些弧。通过对n的完全归纳证明，该图具有2^{n-1}种不同的深度优先搜索森林。提示：记住对$i \geqslant 0$有$1+1+2+4+8+\cdots+2^i=2^{i+1}$是能帮上忙的。

(7) * 假设从图G开始，并为G添加一个新节点x，它是原来的图G中所有节点的前导。如果从节点x开始对新图运行图9-31中的`dfsForest`，就只得到一棵树。如果接着将x从该树中删除，就可以得到若干棵树。这些树与原图G的深度优先搜索森林之间有什么联系？

(8) ** 假设有一幅有向图G，我们已经通过图9-31所示的算法从这幅有向图的表示构建了深度优先生成森林F。现在将弧$u \to v$添加到图G中形成新图H，除了节点v出现在节点u相应邻接表中的某个位置，图H的表示与图G的表示如出一辙。如果现在对H的这种表示运行图9-31所示的算法，在什么条件下会构建出相同的深度优先森林F？也就是说，H的树向弧什么时候会刚好与G的树向弧相同？

9.7 深度优先搜索的一些用途

在本节中，我们会看到如何利用深度优先搜索快速解决一些问题。就像之前那样，这里也是用n表示图的节点数，并用m表示节点数与弧数间的较大者，特别要指出的是，假设$n \leqslant m$总是成立的。这里介绍的算法对用邻接表表示的图来说都要花$O(m)$的时间。第一个算法可以确定有向图是否为无环的。然后对那些无环图，我们会看到如何找出其节点的拓扑排序（拓扑排序在7.10节讨论过，我们会找个恰当的时机回顾一下其定义）。我们还要展示如何计算图的传递闭包（概念还是见7.10节），以及如何比用9.4中给出的算法更快地找到无向图的连通分支。

9.7.1 有向图中环路的寻找

在对有向图G进行深度优先搜索期间，可以在$O(m)$的时间内为所有节点指定后序编号。回想一下9.6节的内容，我们发现尾部按后序小于等于其头部的弧只有后向弧。只要某图存在后向弧$u \to v$，其中v的后序编号不小于u的后序编号，该图中就一定存在环路，如图9-35所示。这条环路是由从u到v的弧以及树中从v到其子孙u的路径组成的。

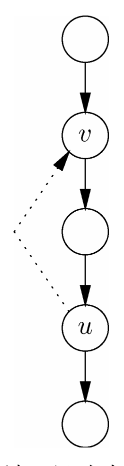


图9-35 每条后向弧都可以与树向弧一起构成环路

这个命题反过来也是成立的，也就是说，如果图中存在环路，那么就肯定存在后向弧。要知道原因，先假设存在某环路，比方说$v_1 \to v_2 \to \cdots \to v_k \to v_1$，并设对$i=1$、2、…、$k$，节点$v_i$的后序编号为$p_i$。如果$k=1$，也就是说，环路为一条弧，那么在图$G$的任意深度优先表示中$v_1 \to v_1$都肯定是后向弧。

如果$k>1$，假设$v_1 \to v_2$，$v_2 \to v_3$，等等，直到$v_{k-1} \to v_k$这些弧都不是后向弧。那么每条弧的头部按后序都先于其尾部，而且后序编号p_1、p_2、…、p_k形成了递减序列。特别要说的是，$p_k < p_1$。然后考虑完成该环路的弧$v_k \to v_1$。其尾部的后序编号为p_k，要小于其头部的后序编号p_1，所以这条弧是后向弧。这就证明了在环路中肯定存在后向弧。

这样一来，在计算了所有节点的后序编号后，只要检查所有弧，看看有没有弧的尾部按后序小于等于其头部。如果有，我们就找到了后向弧，而且该图是有环图。如果没有这样的弧，该图就是无环图。图9-36展示了测试外部定义的图G是否为无环图的函数，它所使用的表示图的数据结构与9.6节中描述的相同。它还利用了图9-33中定义的dfsForest函数计算G中节点的后序编号。

```
        BOOLEAN testAcyclic(GRAPH G)
        {
            NODE u, v; /* u会行经所有的节点 */
            LIST p; /* p指向与u对应的邻接表中的各个单元，
                       v是该邻接表中的节点 */

(1)         dfsForest(G);
(2)         for (u = 0; u < MAX; u++) {
(3)             p = G[u].successors;
(4)             while (p != NULL) {
(5)                 v = p->nodeName;
(6)                 if (G[u].postorder <= G[v].postorder)
(7)                     return FALSE;
(8)                 p = p->next;
                }
            }
(9)         return TRUE;
        }
```

图9-36　确定图G是否无环的函数

在第(1)行调用dfsForest计算后序编号之后，我们会在第(2)行到第(8)行的循环中检查各节点u。指针p会沿着u对应的邻接表向下行进，而且在第(5)行，v会依次成为u的各个后继。如果在第(6)行发现u按后序等于或先于v，就找到了后向弧$u \to v$，并在第(7)行返回FALSE。如果没有找到这样的弧，就在第(9)行返回TRUE。

9.7.2　无环测试的运行时间

和之前一样，设n是图G中节点的数量，并设m是节点数与弧数之间的较大者。我们已经知道在图9-36的第(1)行中对dfsForest的调用要花$O(m)$的时间。而第(5)到第(8)行是while循环的循环体，显然要花$O(1)$的时间。要得到while循环本身运行时间的良好边界，就必须用到我们在9.6节中为深度优先搜索的时间确定边界时用到的小诀窍。设m_u是节点u的出度，那么第(4)行到第(8)行的循环就要进行m_u次。因此，第(4)行到第(8)行所花的时间是$O(1+m_u)$。

第(3)行只要花$O(1)$的时间，因此第(2)行到第(8)行的`for`循环所花的时间是$O(\sum_u(1+m_u))$。正如在9.6节中验证过的，这些1的和是$O(n)$，m_u的和则是m。由于$n \leqslant m$，所以第(2)到第(8)行的`for`循环的运行时间就是$O(m)$。正如深度优先搜索本身那样，检测环路的时间与查看整幅图所花的时间也只有常数因子的差别。

9.7.3　拓扑排序

假设我们知道有向图G是无环的。就像对任意图那样，可以为G找到一个深度优先搜索森林，并为G的节点确定后序编号。假设$(v_1, v_2, \cdots, v_n)$是图G的节点按后序反向排列的表，也就是说v_1是后序编号为n的节点，v_2的编号为$n-1$，而且一般而言，v_i是后序编号为$n-i+1$的节点。

该表中节点的次序具有这样的属性，G中所有弧都是按照该次序向前行进的。要知道原因，假设$v_i \to v_j$是G中的弧。因为G是无环图，所以其中肯定不存在后向弧。因此，对每条弧而言，其头部按后序都是先于尾部的。也就是说，v_j按后序要先于v_i。不过表是与后序反向的，所以在表中v_i要先于v_j。也就是说，按照表的次序每条弧的尾部都要先于其头部。

具有图中每条弧的尾部都先于头部这种属性的图G中，节点的次序叫作*拓扑次序*，而为这些节点找到这一次序的过程就叫作*拓扑排序*。只有无环图才有拓扑次序，而且正如我们已经看到的，通过深度优先搜索，可以在$O(m)$的时间内为无环图生成拓扑次序，其中m是节点数和弧数之间的较大者。如果要为某节点给定后序编号，也就是要完成对该节点的`dfs`调用，我们会将该节点压入栈中。在完成所有调用后，该栈就成了节点按后序出现的表，其中编号最大的节点位于栈顶（前端）。这就是我们所需要的反向后序。因为深度优先搜索要花$O(m)$的时间，而且将节点压入栈只需要$O(n)$的时间，所以整个过程要花费$O(m)$的时间。

拓扑次序和环路查找的应用

本节中讨论的算法在不少情况下是很实用的。当执行用节点表示的特定任务所依照的次序有约束时，拓扑排序就会变得很方便。如果在执行任务v之前必须执行u，就画一条从u到v的弧，然后拓扑次序就是我们执行所有任务所依照的次序。

非递归函数集合的调用图也是相似的例子，在这种情况下我们要在分析了某函数调用的函数之后再分析该函数。因为弧是从调用者到被调用函数的，所以拓扑次序的相反次序，也就是后序本身是我们分析函数所依照的次序，以确保我们只会在处理完某函数调用的函数后再来处理该函数。

在其他情况下，运行该环路测试也是有效的。例如，表示任务优先级的图中所含的环路就表示执行所有任务是没有次序可依照的，而调用图中的环路表示存在递归调用。

✦ 示例 9.22

在图9-37a中有一幅无环图，而在图9-37b中是按照字母表次序考虑节点后得到的深度优先搜索森林。我们在图9-37b中还展示了从这次深度优先搜索中得到的后序编号。如果最先把后序编号最高的节点列出来，我们就得到拓扑次序(d, e, c, f, b, a)。读者应该自行验证一下图9-37a的8条弧中每条弧的尾部按照该表的次序都先于其头部。而这幅图恰好还有另外3种拓扑次序，比如(d, c, e, b, f, a)。

9.7.4　可达性问题

对于有向图可以提出一个很自然的问题：给定一个节点u，沿着有向图中的弧从u可以到达哪些节点？我们将该节点集合称为节点u的可达集。其实，如果对图中每个节点u都询问这一可达性问题，就能知道对哪些节点对(u, v)而言存在从u到v的路径。

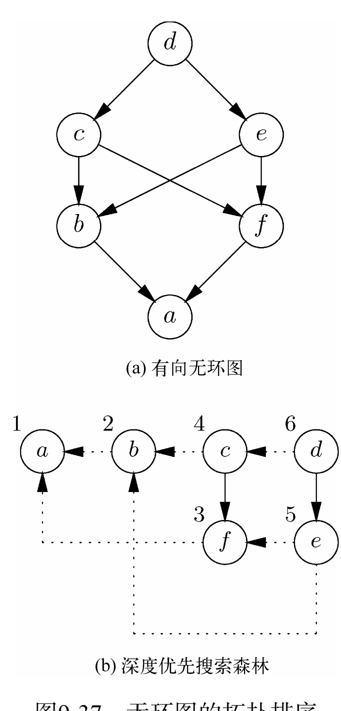
解决可达性问题的算法很简单。如果我们对节点u感兴趣，就将所有节点标记为`UNVISITED`并调用`dfs(u)`。然后可以再次检测所有节点。那些标记上`VISITED`的节点就是从u可达的节点，而其他节点则不是。如果想找到从另一个节点u可达的节点，就将所有节点再次置为`UNVISITED`，并调用`dfs(u)`。我们可以为所有想要处理的节点重复该过程。

✦　示例 9.23

考虑图9-37a。如果从节点a开始进行深度优先搜索，我们哪里都去不了，因为没有从a发出的弧。因此，`dfs(a)`会立即终止。因为只有a被访问过，所以可以得出结论，a是从a可达的唯一节点。

如果从b开始，可以到达a，但这也就是全部了，所以b的可达集就是$\{a, b\}$。同样，从c开始可以到达$\{a, b, c, f\}$，从d则可以到达所有节点，从e可以到达$\{a, b, e, f\}$，而从f只可以到达$\{a, f\}$。

再举个例子，考虑一下图9-26。从a可以到达所有的节点。不过从除了a之外的任意节点出发，可以到达除了a之外的所有节点。



(a) 有向无环图

(b) 深度优先搜索森林

图9-37　无环图的拓扑排序

9.7.5 可达性测试的运行时间

假设有向图G含n个节点与m条弧。还假设G是用9.6节中的`GRAPH`数据类型表示的。首先，假设我们想为某节点u找到其可达集。将所有节点初始化为`UNVISITED`需要$O(n)$的时间。对`dfs(u,G)`的调用要花$O(m)$的时间，而且再次检查所有节点看哪些节点被访问过需要$O(n)$的时间。在检查节点之时，我们还可以创建由从节点u可达的节点组成的表，这仍然只用花$O(n)$的时间。因此，为一个节点找到可达集要花$O(m)$的时间。

现在假设需要为所有n个节点找出可达集。我们可以重复该算法n次，为每个节点执行一次该算法。因此，总时间为$O(mn)$。

9.7.6 通过深度优先搜索寻找连通分支

在9.4节中，我们给出了为节点数为n且节点数与边数较大值为m的无向图寻找连通分支的算法，该算法的运行时间为$O(m \log n)$。我们用于合并分支的树结构本身是很有意义的，例如，可以利用它帮助实现克鲁斯卡尔的最小生成树算法。不过，如果使用深度优先搜索，我们可以更高效地找到连通分支。正如接下来将要看到的，$O(m)$的时间就足够了。

思路就是把无向图当作边被两个方向上的弧替代后的有向图。如果用邻接表表示图，那么甚至不用对这种表示进行任何修改。现在为该有向图构建深度优先搜索森林，森林中的每棵树都是该无向图的一个连通分支。

传递闭包和自反传递闭包

设R是集合S上的二元关系。可达性问题就可以视作计算R的自反传递闭包，通常将其表示为R^*。关系R^*是满足以下条件的有序对$(u，v)$的集合，在由R表示的图中，从节点u到节点v存在长度为0或0以上的路径。

另一种非常类似的关系是R^+，也就是R的传递闭包，它是满足如下条件的有序对$(u，v)$的集合。在由R表示的图中，从节点u到节点v存在长度为1或1以上的路径。R^*和R^+之间的区别在于，$(u，u)$对S中的每个u而言都在R^*中，而当且仅当从u到u之间存在一条长度为1或1以上的环路时，$(u，u)$才在R^+中。要从R^*计算R^+，只需要检查每个节点u是否有来自其可达节点（包括它本身）的进入弧（entering arc），如果没有，就将u从它自己的可达集中删除。

要知道原因，首先要注意到有向图中的弧$u \to v$的出现表示存在边$\{u，v\}$。因此，树中的所有节点都是连通的。

现在我们必须证明反向的命题，也就是如果两个节点是连通的，那么它们在同一棵树中。假设在无向图中存在连通不同树中两个节点u和v的路径。假如u的树是首先构建起来的。那么在有向图中存在从u到v的路径，这表明v和该路径上的所有节点都应该已经被添加到含u的树中。因此，当且仅当无向图中的节点在同一棵树中时，它们才是连通的，也就是说，这些树都是连通分支。

✦ 示例 9.24

再次考虑图9-4中的无向图。图9-38展示了我们可以为该图构建的一种可能的深度优先搜索森林。要注意这3棵深度优先搜索树是如何与3个连通分支对应的。

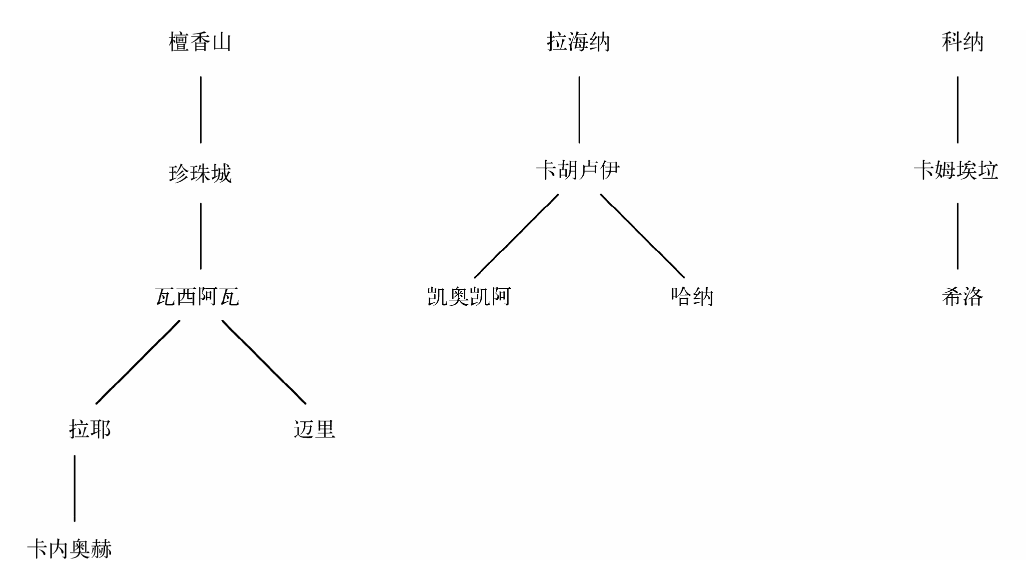


图9-38　深度优先搜索森林将无向图分为连通分支

9.7.7　习题

(1) 为图9-37所示的图找出所有拓扑次序。

(2) * 假设R是定义域D上偏序关系。我们可以用R的图来表示R，其中节点是D中的元素，而且只要uRv且$u \neq v$就存在弧$u \to v$。设$(v_1, v_2, \cdots, v_n)$是R的图的拓扑次序。设关系T是这样定义的，只要$i \leqslant j$，就有v_iTv_j。证明下列命题。

(a) T是全序关系。

(b) R中的有序对是T中有序对的子集，也就是，T是包含了偏序关系R的全序关系。

(3) 对图9-21所示的图（在将其转换为对称的有向图之后）应用深度优先搜索，找出其连通分支。

(4) 考虑含有弧$a \to c$，$b \to a$，$b \to c$，$d \to a$和$e \to c$的图。

(a) 对该图进行环路测试。

(b) 为该图找出所有的拓扑次序。

(c) 为每个节点找出可达集。

(5) * 在9.8节中，我们会考虑找到从源节点s出发的最短路径的一般问题。也就是说，如果从s到各个节点u的最短路径存在，我们希望弄清这些最短路径的长度。如果是有向无环图，这个问题就要简单一些。给出算法，计算有向无环图G中从节点s到各节点u的最短路径的长度，如果这样的路径不存在则是无限长。大家设计的算法应花费$O(m)$的时间，其中m是图G中节点数和弧数的较大者。证明自己的算法具有该运行时间。提示：首先对G进行拓扑排序，依次访问每个节点。在访问节点u时，根据已经计算出的s到u的前导的最短距离来计算从s到u的最短距离。

(6) * 为有向无环图G给出计算以下几项内容的算法。大家给出的算法应该有$O(m)$的运行时间，其中m是G中节点数和弧数的较大者，而且大家应该证明这一运行时间就是自己设计的算法所需要的。提示：对习题(5)中的思路加以改造。

(a) 对每个节点u，找到从u到任意节点的最长路径的长度。

(b) 对每个节点u，找到从任意节点到u的最长路径的长度。

(c) 对给定的源节点s和G的所有节点u，找到从s到u的最长路径的长度。

(d) 对给定的源节点s和G的所有节点u，找到从u到s的最长路径的长度。

(e) 对每个节点u，找到经过u的最长路径的长度。

9.8 用于寻找最短路径的迪杰斯特拉算法

假设有一幅图，它既可能是有向图也可能是无向图，它的弧（或边）都带有表示弧（或边）的“长度”的标号。图9-4就是个例子，它展示了夏威夷群岛的特定公路的距离。想知道两个节点间的最小距离是特别平常的事，例如，地图上通常会带有行车距离的表格，从而指示出人们一天中可以开多远，或是有助于确定经过不同的中间城市的两条路线中哪条更短。类似的问题会给每条弧关联上沿着该弧行进所花的时间，或者可能关联上经过该弧的开销。那么两个节点间的最小“距离”就分别对应着出行的时间或费用。

一般而言，路径的*距离*是该路径上所有弧（或边的）标号之和。而从*节点u到节点v的最小距离*是从u到v的任意路径的距离中最小的那个。

✦ 示例 9.25

考虑一下图9-10中瓦胡岛的地图。假设想要找出从迈里到卡内奥赫的最小距离，有若干路径可供我们选择。有一个实用的观点，只要弧的标号是非负的，那么最小距离路径中就绝对不会有环路。我们可以跳过环路，并在同样的两个节点间找到一条距离不超过含环路路径距离的路径。因此，我们只需要考虑如下路径。

(1) 经过珍珠城和檀香山的路径。

(2) 经过瓦西阿瓦、珍珠城和檀香山的路径。

(3) 经过瓦西阿瓦和拉耶的路径。

(4) 经过珍珠城、瓦西阿瓦和拉耶的路径。

这些路径的距离分别是44、51、67和84。因此，从迈里到卡内奥赫的最小距离是44。

如果想找到从某给定节点（称为*源节点*）到图中所有节点的最小距离，可以使用的最有效的技巧之一就是*迪杰斯特拉算法*，这也就是本节的主题。事实证明，如果我们想要的就是从一个节点u到另一个节点v的距离，最佳方式就是以u为源节点运行迪杰斯特拉算法，并在得出到v的距离时停止算法。如果我们想找到每个节点对之间的最小距离，就要用到9.9节中将要介绍的算法，弗洛伊德算法，该算法有时要比以每个节点为源节点运行迪杰斯特拉算法更值得选择。

迪杰斯特拉算法的本质就是，我们按照这些最小距离从小到大的次序，也就是最近的节点最先的次序，找到从源节点到其他节点的最小距离。随着迪杰斯特拉算法的进行，就有了类似图9-39所示的情形。在图G中，存在某些特定的节点是*已解决的*，也就是说，它们的最小距离是已知的，这一集合总包括源节点s。对未解决的节点v，我们要记录最短*特殊路径*的长度，所谓特殊路径，就是从源节点出发，只经过已解决节点，然后在最后一步跳出已解决区域直接到达v的路径。

我们要为每个节点u记录$dist(u)$值。如果u是已解决节点，那么$dist(u)$就是从源节点到u的最短路径的长度。如果u不是已解决的，那么$dist(u)$就是从源节点到u的最短特殊路径的长度。一开始，只有源节点是已解决的，而且$dist(s)=0$，因为只由s自己构成的路径的长度肯定是0。如果存在从s到u的弧，那么$dist(u)$就是该弧的标号。请注意，在只有s是已解决节点时，特殊路径只包含那些从s出发的弧，所以如果存在从s到u的弧，$dist(u)$就应该是弧$s \to u$的标号。我们还将使用定义的常量`INFTY`，该常量要比图G中任意路径的距离都要大。`INFTY`是作为“无限的”值使用的，表示尚未发现特殊路径。也就是说，如果一开始不存在弧$s \to u$，就有$dist(u)$ = `INFTY`。

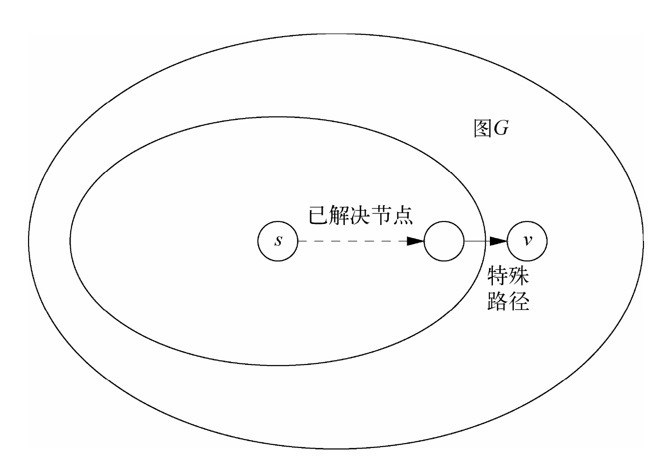


图9-39　迪杰斯特拉算法执行过程中的中间阶段

现在假设我们有些已解决节点和一些未解决节点，如图9-39所示。我们发现节点v是未解决的，但在所有未解决节点中具有最小的*dist*值。可以通过以下方式“解决”v。

(1) 接受$dist(v)$作为从s到v的最小距离。

(2) 对所有尚未解决的节点u，调整$dist(u)$的值，以表明v现在已解决这一事实。

第(2)步所要求的调整如下所述。我们要对$dist(u)$的旧值与$dist(v)$加上弧$v \to u$的标号的和进行比较，如果后者所述的和较小，就将$dist(u)$替换为该和。如果不存在弧$v \to u$，就不用调整$dist(u)$。

✦　示例 9.26

考虑一下图9-10中的瓦胡岛地图。该图是无向图，不过可以假设图中的边是两个方向上的弧。设源节点为檀香山。那么一开始，只有檀香山是已解决的，而且其距离为0。我们可以将*dist*（珍珠城）置为13并将*dist*（卡内奥赫）置为11，不过对其他城市来说，没有从檀香山出发到这些城市的弧，所以它们与檀香山的距离就是`INFTY`。这种情形如图9-40的第一列所示。距离值上的星号表示该节点是已解决的。

城　市	轮　次				
	(1)	(2)	(3)	(4)	(5)
檀香山	0*	0*	0*	0*	0*
珍珠城	13	13	13*	13*	13*
迈里	INFTY	INFTY	33	33	33*
瓦西阿瓦	INFTY	INFTY	25	25*	25*
拉耶	INFTY	35	35	35	35
卡内奥赫	11	11*	11*	11*	11*

*dist*的值

图9-40　迪杰斯特拉算法执行过程中的各个阶段

在这些未解决节点中，具有最小距离的节点现在为卡内奥赫，所以这节点就是已解决的。存在从卡内奥赫到檀香山和拉耶的弧。到檀香山的弧是派不上用场的，不过*dist*（卡内奥赫）的值11，加上从卡内奥赫到拉耶的弧的标号24，总共为35，要小于当前*dist*（拉耶）的值——“无限大”。因此，在第二列中，我们已经把到拉耶的距离减小到35。而卡内奥赫现在是已解决节点了。

在下一轮处理中，具最小距离的未解决节点是珍珠城，其距离为13。在我们解决珍珠城时，还必须考虑珍珠城的邻居，也就是迈里和瓦西阿瓦。我们可以将到迈里的距离减至33（13和20的和），并将到瓦西阿瓦的距离减至25（13和12的和）。现在的情况就如第(3)列所示。

接下来要解决的是瓦西阿瓦，其距离为25，在当前的未解决节点中是最小的。不过，该节点不允许我们减少到其他节点的距离，所以第(4)列与第(3)列有着相同的距离项。同样，接着要解决迈里，其距离为33，不过它也不能减少任何距离，所以第(5)列的各距离项也与第(4)列的相同。严格地说，必须解决最后一个节点，拉耶，不过最后一个节点不可能影响到其他距离，所以第(5)列就给出了从檀香山到所有6个城市的最短距离。

9.8.1 迪杰斯特拉算法起效的原因

为了证明迪杰斯特拉算法是起作用的，必须假设弧的标号都是非负的。[①]我们将通过对k的归纳证明，在存在k个已解决节点时，下列命题成立。

(a) 对每个已解决节点u来说，$dist(u)$是从s到u的最小距离，而且到u的最短路径只由已解决节点组成。

(b) 对每个未解决节点u来说，$dist(u)$是从s到u的任意特殊路径的最小距离，如果不存在这样的路径则为`INFTY`。

依据。对$k = 1$，s是唯一的已解决节点。我们将$dist(s)$初始化为0，这满足了(a)。而对其他每个节点u，如果弧$s \to u$存在，我们就将$dist(u)$初始化为该弧的标号，如果不存在就初始化为`INFTY`。因此，(b)也得到满足。

归纳。现在假设(a)和(b)在k个节点被解决后还成立，并设v是第$k+1$个被解决的节点。我们声明(a)仍然成立，因为$dist(v)$是从s到v的任意路径的最短距离。假设不是，根据归纳假设的(b)部分，当有k个节点已解决时，$dist(v)$是到v的任意路径的最小距离，而且一定存在到距离更短的到v的非特殊路径。如图9-41所示，这一路径一定会在某个节点w（可能是节点s）离开已解决节点，并行向某个未解决节点u。从那里开始，该路径可能反复进出已解决节点，直到它最终到达节点v。

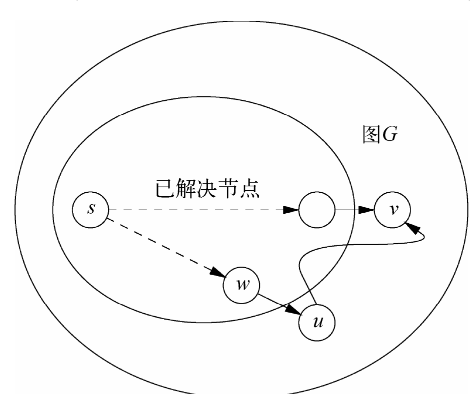


图9-41 假设的经过w和u到达v的更短路径

不过，v被选定为第k+1个已解决节点。这就意味着这时的$dist(u)$不会小于$dist(v)$，否则就要选择u作为第k+1个节点。根据归纳假设的(b)部分，$dist(u)$是到u的任意特殊路径的最小距离。不过图9-41中从s到w再到u的路径是条特殊路径，所以该路径的距离至少为$dist(u)$。因此，假设的

① 如果允许标号是负值，我们就可以找到一些迪杰斯特拉算法会给出错误答案的图。

经过w和u的从s到v的更短路径的距离至少为$dist(v)$，因为该路径从s到u那部分的距离已经是$dist(u)$了，而且有$dist(u) \geqslant dist(v)$。[1]因此，(a)对$k$+1个节点也是成立的，也就是说，当我们将$v$纳入已解决节点的行列时，(a)继续成立。

现在必须证明当把v添加到已解决节点中时，(b)仍然成立。考虑当把v添加到已解决节点中时仍然未解决的某个节点u。在到u的最短特殊路径上，肯定存在某倒数第二个节点，这个节点既可能是v也可能是其他某个节点w。这两种可能如图9-42所示。

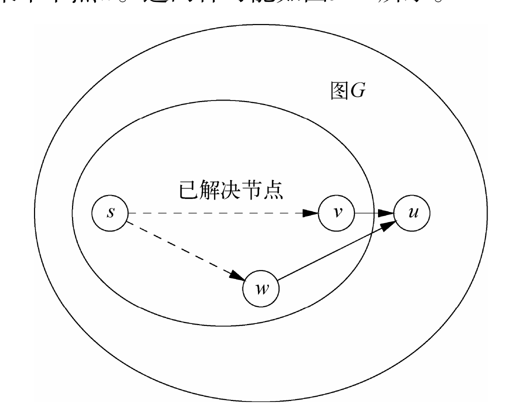


图9-42　到u的最短特殊路径上倒数第二个节点是什么

首先假设倒数第二个节点是v。那么图9-42中所示从s到v再到u的路径的长度就是$dist(v)$加上弧$v \to u$的标号。

另外，假设倒数第二个节点是其他某节点w。根据归纳假设(a)，从s到w的最短路径只由v之前的已解决节点组成，因此，v没有出现在这条路径上。所以，当把v添加到已解决节点中时，到u的最短特殊路径不会改变。

现在回想一下，当解决v时，会把每个$dist(u)$值调整为$dist(u)$的旧值与$dist(v)$加上弧$v \to u$的标号这两者中的较小者。前者表示的是除v之外的某个节点w作为倒数第二个节点的情况，而后者则是v为倒数第二个节点的情况。因此，(b)部分也成立，这样就完成了归纳步骤。

9.8.2　迪杰斯特拉算法的数据结构

现在要展示迪杰斯特拉算法的一种高效实现，它利用了5.9节的平衡偏序树结构。[2]这里要使用两个数组，一个是表示该图的`graph`数组，另一个是表示偏序树的`potNodes`。这样做的目的是，对图中的各节点u，都有一个与之对应的偏序树节点a，其优先级就等于$dist(u)$。不过，和5.9节不同的是，我们将按照最低优先级而不是最高优先级来组织该偏序树。或者，我们可以取$-dist(u)$为a的优先级。图9-43展示了这种数据结构。

我们用`NODE`作为图节点的类型。和往常一样，将用从0开始的整数为节点命名，还将使用`POTNODE`作为偏序树中节点的类型。就像在5.9节中那样，为了方便，我们会假设偏序树中的节点是用从1开始的整数编号的。因此，`NODE`和`POTNODE`类型就等同于`int`。

---

① 请注意，所有标号都非负的事实是至关重要的，如果不这样的话，该路径从u到v的部分的距离有可能为负值，这样就会得到一条到v的更短路径。

② 其实，当弧的数量要比节点数的平方（也就是能达到的弧的最大数量）小一些时，这种实现是唯一的好方法。我们将在习题中讨论用于稠密图情况的一种简单实现。

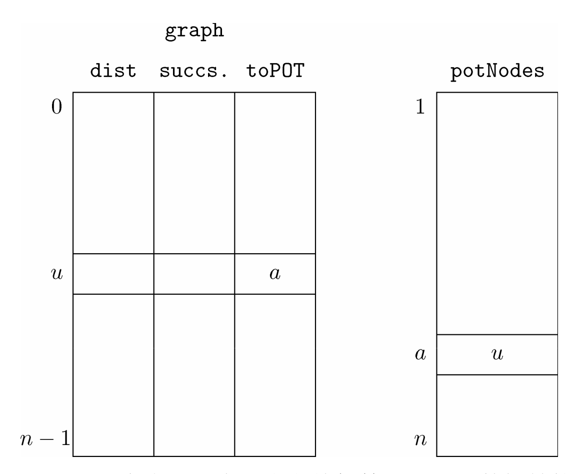


图9-43 用来表示对应迪杰斯特拉算法的图的数据结构

GRAPH数据类型定义如下

```
typedef struct {
    float dist;
    LIST successors;
    POTNODE toPOT;
} GRAPH[MAX];
```

这里，MAX是图中的节点数，而LIST则是由CELL类型的单元组成的邻接表的类型。因为需要加上为浮点数标号的标签，所以就应该像下面这样定义CELL类型

```
typedef struct CELL *LIST;
struct CELL {
    NODE nodeName;
    float nodeLabel;
    LIST next;
};
```

我们将数据类型POT声明为图中节点组成的数组

```
typedef NODE POT[MAX+1];
```

我们现在可以定义以下关键数据结构

```
GRAPH graph;
POT potNodes;
POTNODE last;
```

结构体数组graph含有图中的节点，而数组potNodes则包含了偏序树中的节点，而变量*last*则表示偏序树（存放在数组potNodes[1..last]中）当前的末端。

直觉上讲，偏序树的结构是用数组potNodes中的位置表示的，就像平常表示偏序树那样。该数组中的元素让我们通过引用回图本身来区分节点的优先级。特别要说的是，在potNodes[a]中放置的是所表示图节点的索引u。而dist字段graph[u].dist给出了节点a在偏序树中的优先级。

9.8.3 迪杰斯特拉算法的辅助函数

我们需要若干辅助函数来让实现运转起来。最基础的函数是swap，它会交换偏序树中的两个节点。这个问题并不像5.9节中的那么简单。在这里，graph的toPOT字段必须继续记录数组potNodes中的值，如图9-43所示。也就是说，如果graph[u].toPOT的值为a，那么还肯定有potNodes[a]的值为u。

swap函数的代码如图9-44所示。它接受的参数包括图G、偏序树P，以及偏序树中的两个节点a和b。这里将检验该函数交换偏序树*a*和*b*两项中的值并交换对应图节点的toPOT字段的工作留给读者作为练习。

```
void swap(POTNODE a, POTNODE b, GRAPH G, POT P)
{
    NODE temp; /* 用来交换偏序树节点 */

    temp = P[b];
    P[b] = P[a];
    P[a] = temp;
    G[P[a]].toPOT = a;
    G[P[b]].toPOT = b;
}
```

图9-44　交换偏序树中两个节点的函数

我们将需要让节点在偏序树中上冒与下沉，就像在5.9节中所做的那样。其主要区别在于，这里的数组potNodes中元素的值不是优先级。这个值会把我们带到graph数组中的节点，而在表示该节点的结构体中可以找到字段dist，它会告诉我们优先级。因此我们还需要辅助函数priority返回合适节点对应的dist。我们还将在本节内容中作出如下假设，较小的优先级会上升到偏序树的顶端，而非5.9节中那样是较大优先级。

图9-45展示了priority、bubbleUp和bubbleDown函数，它们都是对5.9节中同名函数进行简单修改后得到的。这些函数都接受图G和偏序树P作为参数。而函数bubbleDown还需要整数参数last，用来表示数组P中当前偏序树的末端。

```
float priority(POTNODE a, GRAPH G, POT P)
{
    return G[P[a]].dist;
}

void bubbleUp(POTNODE a, GRAPH G, POT P)
{
    if ((a > 1) &&
           (priority(a, G, P) < priority(a/2, G, P))) {
        swap(a, a/2, G, P);
        bubbleUp(a/2, G, P);
    }
}

void bubbleDown(POTNODE a, GRAPH G, POT P, int last)
{
    POTNODE child;

    child = 2*a;
    if (child < last &&
           priority(child+1, G, P) < priority(child, G, P))
        ++child;
    if (child <= last &&
           priority(a, G, P) > priority(child, G, P)) {
        swap(a, child, G, P);
        bubbleDown(child, G, P, last);
    }
}
```

图9-45　偏序树中节点的上冒与下沉

9.8.4 初始化

我们将假设对应图中各节点的邻接表已经创建，而且指向图节点u的邻接表的指针已经出现在`graph[u].successors`中。还要假设节点0是源节点。如果接受图节点i与偏序树节点i+1对应，数组`potNodes`就恰如其分地被初始化为偏序树。也就是说，偏序树的根节点表示图的源节点，也就是我们给定优先级0的节点，而且我们要为所有其他节点给定优先级`INFTY`，这是定义为“无限”的常量。

带异常的初始化

请注意，在图9-46的第(2)行中，我们将`dist[1]`以及所有其他距离都置为`INFTY`。接着在第(5)行，再将该距离修正为0。与测试每个i值以确定其是否为异常情况相比，这种方式要更具效率。诚然，如果我们将第(2)行替代为

```
if (i == 0)
    G[i].dist = 0;
else
    G[i].dist = INFTY;
```

就可以消除第(5)行，不过这样一来不仅增加了代码，还会增加运行时间，因为这样修改就必须进行n次测试和n次赋值，而不是像图9-46中的第(2)行和第(5)行那样只用进行n+1次赋值而不用进行测试。

正如我们将要看到的，在迪杰斯特拉算法的第一轮处理中，我们选择“解决”源节点，这样就创造了可以作为非正式介绍中起始点的条件，其中源节点是已解决的，而且`dist[u]`只有在存在从源节点到u的弧时才不是无限长。初始化函数如图9-46所示。正如本节中之前的函数那样，`initialize`接受图和偏序树作为参数，还要接受指向整数`last`的指针`pLast`，所以该函数可以将`pLast`初始化为*MAX*，也就是该图中节点的数量。回想一下，`last`表示的是与当前正使用的偏序树对应的数组中最后一个位置。

```
        void initialize(GRAPH G, POT P, int *pLast);
        {
            int i; /* i既作为图节点，也作为树节点，*/

(1)         for (i = 0; i < MAX; i++) {
(2)             G[i].dist = INFTY;
(3)             G[i].toPOT = i+1;
(4)             P[i+1] = i;
            }
(5)         G[0].dist = 0;
(6)         (*pLast) = MAX;
        }
```

图9-46 迪杰斯特拉算法的初始化

请注意，偏序树的索引是1到*MAX*，而对图来说，这些索引就是从0到*MAX*–1。因此，在图9-46的第(3)行和第(4)行中，必须一开始就把图中的节点i对应到偏序树的节点i+1。

9.8.5 迪杰斯特拉算法的实现

图9-47展示了迪杰斯特拉算法的代码，利用到了我们之前已经编写的所有函数。要为对应偏序树potNodes以及带有指示偏序树末端的整数last的图graph执行迪杰斯特拉算法，就要初始化这些变量，然后调用

```
Dijkstra(graph, potNodes, &last)
```

函数Dijkstra的工作原理如下。在第(1)行我们要调用initialize。其余代码，也就是第(2)行到第(13)行是个循环，该循环每次迭代都对应迪杰斯特拉算法的一轮处理，在迭代中我们要选择一个节点v并将其解决。在第(3)行选择的节点v总是对应树节点为偏序树根节点的那个。在第(4)行，通过交换v与偏序树当前的最后一个节点，我们把v从偏序树中取出。第(5)行其实是通过递减last将v删除。然后第(6)行通过对我们刚放置到根节点位置的节点调用bubbleDown，还原了偏序树属性。事实上，未解决节点会出现在*last*之下，而已解决节点则出现在*last*及*last*以上的位置。

```
     void Dijkstra(GRAPH G, POT P, int *pLast)
     {
         NODE u, v; /* v 是我们要解决的节点 */
         LIST ps; /* ps 沿着v的后继表下行，u是ps
                     指向的那个后继 */

(1)      initialize(G, P, pLast);
(2)      while ((*pLast) > 1) {
(3)          v = P[1];
(4)          swap(1, *pLast, G, P);
(5)          --(*pLast);
(6)          bubbleDown(1, G, P, *pLast);
(7)          ps = G[v].successors;
(8)          while (ps != NULL) {
(9)              u = ps->nodeName;
(10)             if (G[u].dist > G[v].dist + ps->nodeLabel) {
(11)                 G[u].dist = G[v].dist + ps->nodeLabel;
(12)                 bubbleUp(G[u].toPOT, G, P);
                 }
(13)             ps = ps->next;
             }
         }
     }
```

图9-47　迪杰斯特拉算法的主函数

在第(7)行我们开始调整距离，以反映v现在已被解决的事实。指针p被初始化为指向节点v邻接表的开头。在第(9)行把变量u置为v的某一后继之后，我们在第(10)行测试了到u的最短特殊路径是否经过v。只要该数据结构中由G[u].dist表示的$dist(u)$的旧值大于$dist(v)$加上弧$v \to u$标号的和，就有这一最短特殊路径经过节点v。如果这样，那么在第(11)行，将$dist(u)$置为新的更小的值，并在第(12)行调用bubbleUp。所以，如果需要的话，u可以在偏序树中上升到反映其新优先级的位置。在第(13)行中随着p沿着v的邻接表向下移动，该循环就随之完成了。

9.8.6 迪杰斯特拉算法的运行时间

正如之前所述那样，假设图具有n个节点，而且m是弧数与节点数的较大者。这里将会按照描述这些函数的次序分析每个函数的运行时间。首先，swap函数显然花费$O(1)$的时间，因为它

只由赋值语句组成。同样`priority`函数也花费$O(1)$的时间。

`bubbleUp`函数是递归函数，但它的运行时间是$O(1)$加上对到根节点一半距离的节点递归调用该函数的时间。正如我们在5.9节中验证过的，至多存在$\log n$次调用，而且每次调用需要$O(1)$的时间，所以`bubbleUp`的总运行时间就是$O(\log n)$。同样，`bubbleDown`也花费$O(\log n)$的时间。

`initialize`函数要花$O(n)$的时间。具体来说就是，第(1)行到第(4)行的循环要迭代n次，而其循环体每次迭代要花$O(1)$的时间。这就给了该循环$O(n)$的时间。第(5)行和第(6)行各自贡献了$O(1)$的时间，不过我们在大O表达式中可以将其省略掉。

现在把注意力转移到图9-47中的`Dijkstra`函数上。设m_v是节点v的出度，或者说是v的邻接表的长度。我们首先分析第(8)行到第(13)行的内层循环。第(9)行到第(13)行的各行都要花$O(1)$的时间，除了第(12)行对`bubbleUp`的调用之外，我们论证过该调用要花$O(\log n)$的时间。因此，该循环的循环体要花$O(\log n)$的时间。该循环进行的次数等于v对应的邻接表的长度，之前已经将其表示为m_v了。因此，第(8)行到第(13)行的循环的运行时间可以表示为$O(1+m_v \log n)$，1这一项表示的是v没有后继，也就是$m_v = 0$的情况，不过我们还是要进行第(8)行的测试。

现在考虑第(2)行到第(13)行的外层循环。我们已经验证过第(8)行到第(13)行了。第(6)行调用`bubbleDown`需要的时间。而循环体的其他各行都只要O(1)的时间，因此整个循环体需要花上$O((1+m_v) \log n)$的时间。

外层循环刚好要迭代$n-1$次，因为`last`的范围是从n减少到2。$1+m_v$中的1这一项因此要贡献$n-1$，或者说是$O(n)$的时间。不过，m_v这项一定要为各个节点v相加，因为所有节点（除了最后一个节点）都要被选作一次v。因此，m_v对外层循环所有迭代的总贡献是$O(m)$，因为有$\sum_v m_v \leqslant m$。由此可以得出外层循环要花$O(m \log n)$的时间。第(1)行对`initialize`的调用要花$O(n)$的时间，不过可以将其省略。这样一来就得到迪杰斯特拉算法的运行时间为$O(m \log n)$，也就是说，至多是查看图中的节点和弧所花时间的$\log n$倍。

9.8.7　习题

(1) 按照图9-21中所示的图（见9.4节的习题），找到从底特律到其他城市的最短距离。如果某城市是从底特律不可达的，则这个最小距离为“无限”。

(2) 有时候我们希望计算从一个节点到另一个节点遍历的弧的数量。例如，我们可能希望把在乘飞机或坐公交车出行过程中的换乘次数减少到最小。如果给每条弧标记上1，那么最小距离计算就会变成数弧的过程。为图9-5中的图（见9.2节的习题）找出从节点a到达各节点所需要的最小弧数。

(3) 图9-48a中有7个灵长类物种以及它们名称的缩写。这些物种中的某些物种已知要先于其他物种出现，因为在同一地点代表时间流逝的不同地层中发现了它们的化石。图9-48b中的表给出的三元组(x，y，t)表示物种x与物种y是在同一地点被发现，不过x要比y早t百万年出现。

　(a) 画出表示图9-48中数据的有向图，其中弧是从较早的物种到达较晚的物种，弧的标号就表示物种间的时间差异。

　(b) 以AF为源节点，对(a)小题画出的图运行迪杰斯特拉算法，找到AF之后的各其他物种与它相差的最短时间。

(4) * 我们给出的迪杰斯特拉算法的实现需要$O(m \log n)$的时间，这一时间要小于$O(n^2)$的时间，除了弧的数量接近n^2的情况之外。如果m很大，就可以谋划另一种不需要用到优先级队列的实现，这样在每一轮处理中就不再需要$O(n)$的时间选择要解决的节点，而只需要$O(m_u)$的时间，也就是与从已解决节点u出发的弧的数量成正比的时间，来更新$dist$。得到的是一种$O(n^2)$时间的算法。完善这里提出的思路，并为迪杰斯特拉算法的这种实现编写C语言程序。

Australopithecus Afarensis（最早南方古猿）	AF
Australopithecus Africanus（非洲南方古猿）	AA
Homo Habilis（能人）	HH
Australopithecus Robustus（粗壮南方古猿）	AR
Homo Erectus（直立人）	HE
Australopithecus Boisei（鲍氏南方古猿）	AB
Homo Sapiens（智人）	HS

(a) 物种及其缩写

物种1	物种2	时　间
AF	HH	1.0
AF	AA	0.8
HH	HE	1.2
HH	AB	0.5
HH	AR	0.3
AA	AB	0.4
AA	AR	0.6
AB	HS	1.7
HE	HS	0.8

(b) 物种1先于物种2的时间

图9-48　灵长类物种之间的关系

(5) ** 如果某些弧的标号为负值，迪杰斯特拉算法就不是永远都能奏效了。给出一幅能让迪杰斯特拉算法给出错误最小距离的带负值标号的图。

(6) ** 设G是我们已经为其运行过迪杰斯特拉算法并以某种次序解决了其中节点的图。假设要为G添加一条权为0的弧$u \to v$，以形成新图G'。在什么情况下迪杰斯特拉算法为G'解决节点的次序与为G解决节点的次序相同？

(7) ** 在本节中我们采用的方法是把表示图G的数组与存储整数（作为指向其他数组索引）的偏序树关联起来。另一种方式是使用指向数组元素的指针。使用指针替代整数索引，重新实现迪杰斯特拉算法。

9.9　最短路径的弗洛伊德算法

如果想知道含n个节点的非负标号图中所有节点对之间的最小距离，可以把每个节点n作为源节点来运行迪杰斯特拉算法。因为运行一次迪杰斯特拉算法所需的时间为$O(m \log n)$，其中m是节点数和弧数的较大者，那么用这种方式找出所有节点对之间的最小距离就要花掉$O(mn \log n)$的时间。此外，如果m接近其最大值n^2，就可以使用9.8节的习题(4)中讨论过的迪杰斯特拉算法$O(n^2)$时间的实现，这样要为每个节点对都找到最小距离，就需要运行n次成为$O(n^3)$时间的算法。

还有一种找出所有节点对之间最小距离的算法——*弗洛伊德算法*。这种算法要花$O(n^3)$的时间，因此从本质上讲不比迪杰斯特拉算法更好，而且当弧的数量远小于n^2时要比前者更糟。不过，弗洛伊德算法是基于邻接矩阵，而不是基于邻接表的，它从概念上讲要比迪杰斯特拉算法简单得多。

弗洛伊德算法的本质是，依次将图的每个节点u作为*枢纽*。当u是枢纽时，我们会试着利用u

作为所有节点对之间的中间节点，如图9-49所示。对每个节点对（比方说是v和w）而言，如果弧 $v \to u$ 和 $u \to w$ 的标号的和，即图9-49中的$d+e$，要小于当前从v到w的弧的标号f，那么就将f替换为$d+e$。

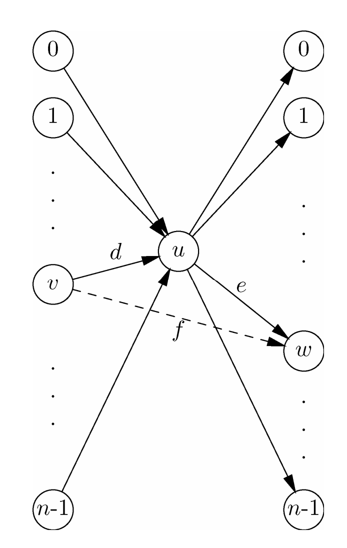


图9-49 使用节点u作为枢纽，以改善某些节点对之间的距离

实现弗洛伊德算法的代码片段如图9-50所示。和之前一样，假设节点使用从0开始的整数命名，并且还是使用NODE作为节点的类型，不过这里要假设该类型为整数或等价的枚举类型。我们还要假设有 $n \times n$ 的数组arc，满足arc[v][w]是给定图中弧 $v \to w$ 的标号。不过，对其对角线上所有的节点v来说，即便存在弧 $v \to v$，也有arc[v][v] = 0。原因在于，从一个节点到其自身的最短距离总是0，而且我们根本不希望沿着这些弧行进。如果没有从v到w的弧，我们就让arc[v][w]为INFTY，就是要比其他任意标号都大很多的一个特殊值。还有一个相似的数组dist，在其末端存放着最小距离，dist[v][w]将成为从节点v到节点w的最小距离。

```
        NODE u, v, w;

(1)     for (v = 0; v < MAX; v++)
(2)         for (w = 0; w < MAX; w++)
(3)             dist[v][w] = arc[v][w];
(4)     for (u = 0; u < MAX; u++)
(5)         for (v = 0; v < MAX; v++)
(6)             for (w = 0; w < MAX; w++)
(7)                 if (dist[v][u] + dist[u][w] < dist[v][w])
(8)                     dist[v][w] = dist[v][u] + dist[u][w];
```

图9-50 弗洛伊德算法

第(1)行到第(3)行会将dist初始化为arc。第(4)行到第(8)行构成了一个循环，其中每个节点u会依次被取作枢纽。对各枢纽u，在v和w上的双重循环中，我们考虑了每个节点对。第(7)行会测试从v经过u到达w是否比从v直接到w更近，如果是这样，那么第(8)行就会将dist[v][w]降低为从v到u的距离及从u到w的距离之和。

✦ 示例 9.27

考虑一下9.3节图9-10中的图。使用0到5的数字表示节点，其中0表示拉耶，1表示卡内奥赫，等等。图9-51展示了arc矩阵，标号INFTY表示相应的节点对之间没有边连通。而arc矩阵也是dist矩阵的初始值。

沃夏尔算法

有时候，我们只对分辨两个节点间是否存在路径感兴趣，而不去管最小距离是多少。如果这样，就可以使用元素类型为BOOLEAN（即int）的邻接矩阵，其中TRUE（1）表示弧的存在，而FALSE（0）表示弧不存在。同样，dist矩阵的元素也是BOOLEAN类型的，其中TRUE表示问题中已知的节点间存在路径，而FALSE表示它们之间不存在路径。我们需要对弗洛伊德算法进行的唯一修改就是把图9-50中的第(7)行和第(8)行替换为

```
(7)  if (dist[v][w] == FALSE)
(8)      dist[v][w] = dist[v][u] && dist[u][w];
```

如果dist[v][w]还不是TRUE，只要dist[v][u]和dist[u][w]都是TRUE，这两行代码就会把它置为TRUE。

得到的算法名为沃夏尔算法（Warshall's Algorithm），可以在$O(n^3)$的时间内为含n个节点的图计算自反闭包和传递闭包。这一算法从来都不会优于9.7节中利用了深度优先搜索的$O(mn)$时间的算法。不过，沃夏尔算法使用的是邻接矩阵而非邻接表，而且如果m接近n^2，它实际上会因为其简单性而比乘法的深度优先搜索更有效率。

请注意，图9-10中的图是无向图，所以矩阵是对称的，也就是说arc[v][w]=arc[w][v]。如果图是有向图，就可能不存在这种对称性，不过弗洛伊德算法并未利用到对称性，因此处理有向图或无向图都是可以的。

	0	1	2	3	4	5
0	0	24	INFTY	INFTY	INFTY	28
1	24	0	11	INFTY	INFTY	INFTY
2	INFTY	11	0	13	INFTY	INFTY
3	INFTY	INFTY	13	0	20	12
4	INFTY	INFTY	INFTY	20	0	15
5	28	INFTY	INFTY	12	15	0

图9-51 arc矩阵，它是dist矩阵的初始值

第一个枢纽是$u=0$。因为INFTY与任意值的和都是INFTY，所以唯一的节点对v和w，两者都不为u。而且有dist[v][u]+dist[u][w]小于INFTY的节点对，就是$v=1$和$w=5$，反之亦然。[①]因为此时dist[1][5]是INFTY，我们就将dist[1][5]用dist[1][0]+dist[0][5]的和52替换掉。同样，要将dist[5][1]替换为52。其他距离都不能借助枢纽0得到改善，这样一来就得到如图9-52所示的dist矩阵。

---

① 如果v和w中有一个为u，就很容易看出dist[v][w]永远不可能借助经过u而得到改进。因此，在查找经过中枢w能够改进距离的节点对时，可以忽略那些形如(v, u)或(u, w)的有序对。

	0	1	2	3	4	5
0	0	24	INFTY	INFTY	INFTY	28
1	24	0	11	INFTY	INFTY	52
2	INFTY	11	0	13	INFTY	INFTY
3	INFTY	INFTY	13	0	20	12
4	INFTY	INFTY	INFTY	20	0	15
5	28	52	INFTY	12	15	0

图9-52 在使用0作为枢纽后的dist矩阵

现在以节点1作为枢纽。在如图9-52所示的当前dist矩阵中，节点1分别存在到节点0（距离24）、节点2（距离11）和节点5（距离52）的非无限连接。我们可以将这些边组合起来，从而将节点0到节点2的距离从INFTY减少到24+11 = 35。还可以把节点2和节点5之间的距离减少到11+52 = 63。请注意，63是从檀香山到卡内奥赫，然后到拉耶，最后到瓦西阿瓦的路径的距离，不过这一最短路线只经过了目前已经成为过枢纽的节点。最终，我们会发现经过珍珠城的更短路线。当前的dist矩阵如图9-53所示。

	0	1	2	3	4	5
0	0	24	35	INFTY	INFTY	28
1	24	0	11	INFTY	INFTY	52
2	35	11	0	13	INFTY	63
3	INFTY	INFTY	13	0	20	12
4	INFTY	INFTY	INFTY	20	0	15
5	28	52	63	12	15	0

图9-53 在使用1作为枢纽后的dist矩阵

现在用2作为枢纽。节点2当前与节点0（距离35）、节点1（距离11）、节点3（距离13）和节点5（距离63）之间存在非无限连接。在这些节点中，节点0和节点3之间的距离可以改善为35+13 = 48，而且节点1和节点3之间的距离可以改善为11+13 = 24。因此，当前的dist矩阵如图9-54所示。

	0	1	2	3	4	5
0	0	24	35	48	INFTY	28
1	24	0	11	24	INFTY	52
2	35	11	0	13	INFTY	63
3	48	24	13	0	20	12
4	INFTY	INFTY	INFTY	20	0	15
5	28	52	63	12	15	0

图9-54 使用节点2作为枢纽后的dist矩阵

接下来，节点3成为枢纽。图9-55展示了节点3与其他各节点之间当前的最佳距离。[①]通过行经节点3，可以作出如下距离上的改善。

(1) 节点1和节点5之间地距离被减小到36。

① 读者应该将图9-55与图9-49加以比较。后者展示了如何在有向图的一般情况下使用枢纽节点，其中进出枢纽节点的弧可能有着不同标号。而图9-55则利用了示例图的对称性，让我们使用节点3与其他各节点之间的边来表示进入节点3的弧，就像图9-49左侧那样，以及从节点3出发的弧，就像图9-49右侧那样。

(2) 节点2和节点5之间的距离被减小到25。
(3) 节点0和节点4之间的距离被减小到68。
(4) 节点1和节点4之间的距离被减小到44。
(5) 节点2和节点4之间的距离被减小到33。

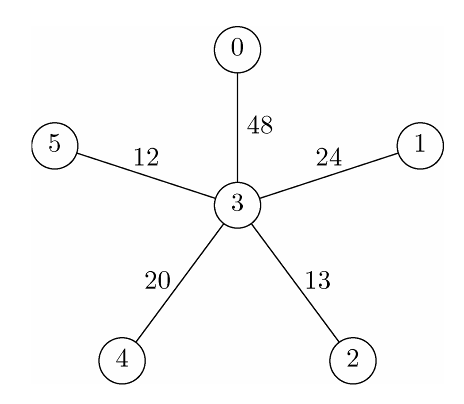


图9-55　到节点3的当前最佳距离

当前的`dist`矩阵如图9-56所示。

	0	1	2	3	4	5
0	0	24	35	48	68	28
1	24	0	11	24	44	36
2	35	11	0	13	33	25
3	48	24	13	0	20	12
4	68	44	33	20	0	15
5	28	36	25	12	15	0

图9-56　使用节点3作为枢纽后的`dist`矩阵

使用节点4作为枢纽不能改善任何距离。当节点5作为枢纽时，我们可以改善节点0和节点3之间的距离，因为在图9-56中有

`dist[0][5]` + `dist[5][3]` = 40

这要小于`dist[0][3]`的值48。就具体的城市而言，这就相当于发现，从拉耶出发经过瓦西阿瓦到珍珠城，要比经过卡内奥赫和檀香山到珍珠城更近。同样，可以把节点0和节点4之间的距离从68改善到43。最终的dist矩阵如图9-57所示。

	0	1	2	3	4	5
0	0	24	35	40	43	28
1	24	0	11	24	44	36
2	35	11	0	13	33	25
3	40	24	13	0	20	12
4	43	44	33	20	0	15
5	28	36	25	12	15	0

图9-57　最终的`dist`矩阵

9.9.1　弗洛伊德算法为何奏效

正如我们所见，在弗洛伊德算法的任何阶段，从节点v到节点w的距离都将是只由做过枢纽的节点组成的路径中最短路径的距离。最终，所有的节点都成为过枢纽，而且`dist[v][w]`存放着所有可能路径的最小距离。

我们可以将从节点v到节点w的*k*路径定义为满足从v到w的中间节点编号不大于k的路径。请注意，并不需要限制v或w一定是k或更小。

$k = -1$的情况是个重要特例。因为假设节点的编号是从0开始的，所以(-1)路径是没有中间节点的。它只可能是一条弧，或是作为0长度路径起止点的一个节点。

图9-58展示了k路径的样子，不过端点v和w既可以在k之上，也可以在k以下。在该图中，线的高度表示了从v到w的路径上各节点的编号。

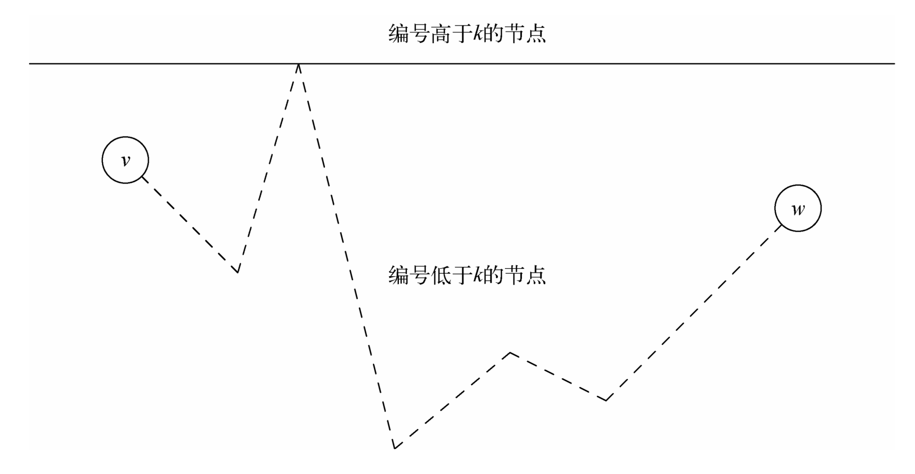


图9-58　除了端点（可能高于k）外，k路径上的节点不会高于k

✦ 示例 9.28

在图9-10中，路径0、1、2、3是一条2路径。中间节点1和2都不大于2。这一路径也是3路径，4路径和5路径。但它不是1路径，因为中间节点2大于1。同样，它也不是0路径或(-1)路径。

因为假设节点的编号是从0到$n-1$，所以(-1)路径不可能有中间节点，因此它一定是一条弧或一个节点。而$n-1$路径则可以是任意路径，因为在节点编号为0到$n-1$的图中，任何路径中间节点的编号都不会大于$n-1$。这里将通过对k的归纳证明该命题。

命题 $S(k)$。如果弧的标号都是非负的，那么在图9-50第(4)行到第(8)行的循环将u置为k+1之前，`dist[v][w]`是从v到w的最短k路径的长度，如果没有这样的路径，其长度就是`INFTY`。

依据。依据是$k = -1$。我们在第一次执行该循环的循环体之前将u置为0。在第(1)行到第(3)行中我们已经把`dist`初始化为`arc`。因为由一个节点构成的弧和路径只有(-1)路径，所以依据成立。

归纳。假设$S(k)$成立，考虑$u = k+1$时循环迭代期间`dist[v][w]`发生的情况。假设P是从v到w的最短k+1路径。有两种情况，具体取决于P是否经过k+1号节点。

(1) 如果P是k路径，也就是说P其实没有经过 k+1号节点，那么根据归纳假设，在经过k次迭代之后`dist[v][w]`已经等于P的长度。因为没有更短的$(k+1)$路径，所以我们在以k+1号节点为枢纽的这轮处理中不能改变`dist[v][w]`。

(2) 如果P是k+1路径，我们可以假设P只经过节点k+1一次，因为环路永远都不可能减少距离。回想一下，我们要求所有标号都是非负的。因此，P是由从v到节点k+1的k路径Q，后面跟上从节点k+1到w的k路径R组成的，如图9-59所示。根据归纳假设，`dist[v][k+1]`和`dist[k+1][w]`分别是在第k次迭代之后路径Q和R的长度。

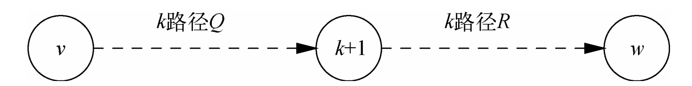


图9-59　(k+1)路径可以分为两条k路径，Q后面跟上R

首先要看到`dist[v][k+1]`和`dist[k+1][w]`在第k+1次迭代中不可能改变。原因在于所有的标号都是非负的，这样所有路径的长度都是非负的，因此图9-50中第(7)行的测试在u（即节点k+1）是v或w其中之一时一定会失败。

所以，当我们以$u = k+1$对任意的v和w应用第(7)行的测试时，自从第k次迭代结束之后`dist[v][k+1]`和`dist[k+1][w]`的值就不再改变。也就是说，第(7)行的测试会对最短k路径的长度及从v到k+1和k+1到w的最短k路径长度之和进行比较。在第(1)种情况下，路径P不经过k+1，前者更短，而在情况(2)中，P经过k+1，后者是图9-59中路径Q和路径R长度之和，因此要更短。

可以得出结论：第k+1次迭代会把`dist[v][w]`置为对所有的节点v和w而言最短k+1路径的长度。这就是命题$S(k+1)$，所以我们得出了归纳的结论。

要完成证明，设$k = n-1$。也就是说，我们知道在完成全部n次迭代后，`dist[v][w]`是任意从v到w的$n-1$路径的最短距离，这样就证明了`dist[v][w]`是从v到w的任意路径中最小的距离。

9.9.2　习题

(1) 假设图9-5（见9.2节习题）中所有的弧标号都为1，使用弗洛伊德算法得出各节点对间最短路径的长度。给出以每个节点作为枢纽后的距离矩阵。

(2) 对图9-5中的图应用沃夏尔算法，计算其自反闭包和传递闭包。给出以每个节点作为枢纽后的可达性矩阵。

(3) 使用弗洛伊德算法为图9-21（见9.4节习题）中的图找到各城市对之间的最短距离。

(4) 使用弗洛伊德算法为图9-48（见9.8节习题）中的各灵长类物种找出它们之间可能间隔的最短时间。

(5) 有时我们想只考虑有一条或多条弧的路径，而将单个节点排除在弧的范畴之外。如何修改`arc`矩阵的初始化部分，使得在查找从节点到其自身的最短路径时只考虑长度为1或1以上的路径？

(6) * 找出图9-10中所有的无环2路径。

(7) * 为什么当弧上的正负开销都存在时弗洛伊德算法就不起作用了？

(8) ** 给出能找到两个给定节点间最长无环路径的算法。

(9) ** 假设对图G运行弗洛伊德算法。然后，我们将弧$u \to v$的标号降为0，以构建新图G'。当对图G和图G'应用弗洛伊德算法时，怎样的节点s和t组成的节点对会使每一轮处理中对应的两个`dist[s][t]`都相同？

9.10　图论简介

图论是专门研究图属性的数学分支。在之前的几节中，我们已经展示了图论的基本定义，以及计算机科学家开发的一些可以高效计算图关键属性的基础算法。我们已经看到计算最短路径、生成树和深度优先搜索树的算法。本节要介绍图论中一些更重要的概念。

9.10.1　完全图

每一对不同的节点之间都存在边的无向图就叫作完全图。含n个节点的完全图就叫作K_n。图9-60展示了从K_1到K_4的完全图。

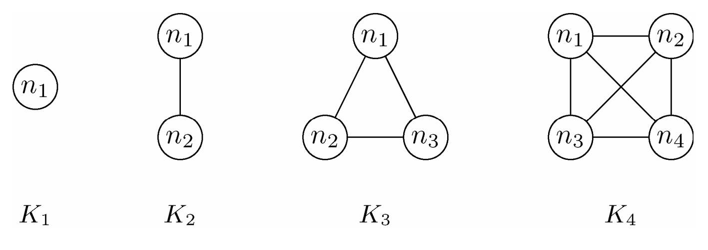


图9-60　前4个完全图

K_n中的边数是$n(n-1)/2$，或者说是$\binom{n}{2}$。想知道原因，可以考虑K_n的边$\{u, v\}$。可以选择n个节点中的任意一个作为u，并从其余$n-1$个节点中任选一个作为v，因此总的选择数为$n(n-1)$。不过，这样一来每条边都数了两次，一次是$\{u, v\}$，第二次是$\{v, u\}$，所以必须在总选择数上除以2，以得出正确的边数。

也存在完全有向图的定义。这种图具有从每个节点到每个其他节点（包括其自身）的弧。n个节点的完全有向图有n^2条弧。图9-61展示了含3个节点和9条弧的完全有向图。

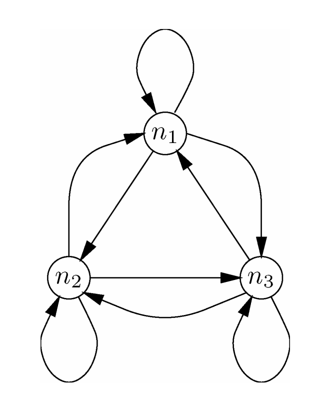


图9-61　含3个节点的完全有向图

9.10.2　平面图

如果可以将无向图的所有节点都放在一个屏幕上，而且可以将它的边画为连续的线而没有边交叉，就说该无向图是平面图。

✦　示例 9.29

图9-60中的K_4在画出来时有两条相交的对角边。不过，K_4是平面图，正如我们看到的图9-62中这种画法一样。在图9-62中，通过重新把一条对角边画在外面，就避免了两条边相交的情况出现。可以说图9-62是图K_4的平面表示，而图9-60则是图K_4的非平面表示。请注意，在平面表示中边可以不是直线。

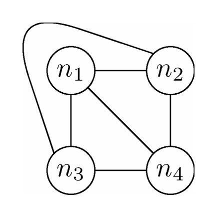


图9-62　K_4的平面表示

在图9-63中看到两种最简单的非平面图，也就是没有任何平面表示的图。一个是K_5，有5个节点的完全图。而另一个是$K_{3,3}$，它的形成方式是，取两组3个节点，并用一组的各个节点与另一组的各个节点连接，但不连接同组中的节点。读者应该试着重画这两种图，看看有没有办法使得任意两条边都不会交叉，以感受一下它们为什么不是平面图。

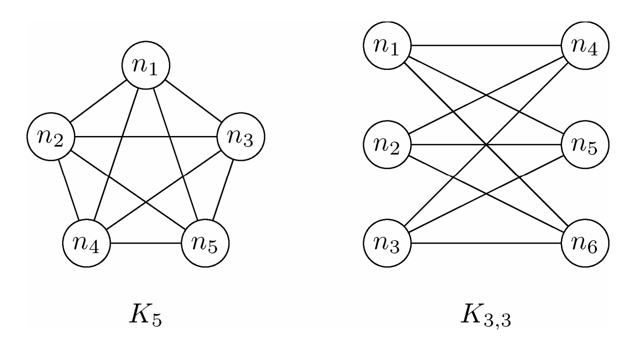


图9-63　两种最简单的非平面图

库拉托斯基定理说过，任何非平面图都至少含有这两种图中一种的“副本”。不过，我们在解释副本的概念时一定要小心一点，因为要在任意的非平面图G中找到K_5或$K_{3,3}$的副本，可能必须要将图9-63所示的某些边与图G中的路径关联起来。

9.10.3　平面性的应用

平面性在计算机科学中是举足轻重的。例如，很多图或相似的图表需要在计算机屏幕或纸上表现出来。为了清楚起见，是需要制出图的平面表示的，或者如果图不是平面图，就需要尽可能减少交叉边的数量。

读者可能会在第13章中看到我们画的一些相当复杂的电路图，它们其实就是以门和线路连接点为节点而且以线路为边的图。因为这些电路一般而言都不是平面的，所以必须制定一些约定，让线路可以在不连接的情况下交叉，而且用点来表示线路的连接。

相关的应用涉及集成电路的设计。集成电路，或者说“芯片”，把第13章中讨论过的那些逻辑电路都实物化了。它们不需要逻辑电路被刻画为平面表示，但存在相似的限制让我们可以为边指定若干“层”，通常是3或4层。在第一层上，表示电路的图必须有平面表示，边是不允许有交叉的。不过，不同层级上的边是可以交叉的。

9.10.4　图着色

为图G着色的图着色问题就是给每个节点指定一种“颜色”，使得由边连通的两个节点不会被指定相同的颜色。然后我们可能要问以这种方式为图着色需要多少种不同的颜色。为图G着

色所需的最小颜色数量就叫作图G的色数（chromatic number），通常表示为$\chi(G)$。可以用不超过k种颜色着色的图被称为可着k色的。

✦ 示例 9.30

如果图是完全图，那么它的色数就等于节点的数量，也就是说$\chi(K_n)=n$。要证明这一点很简单，因为任意两个节点u和v之间都是有边连通的，所以不可能为它们着上相同的颜色。因此，每个节点都要有其自己的颜色。对每个$k \geqslant n$而言，K_n都是可着k色的，不过如果$k<n$，则K_n不是可着k色的。请注意，可以说K_4是可着5色的，即便没法为图K_4的4个节点用上所有5种颜色。然而，严格地讲，只要图可以用k种或更少的颜色着色，而不是说刚好可用k种颜色着色，就可以说图是可着k色的。

再举个例子，图9-63所示的图$K_{3,3}$的色数为2。比方说可以把左边组中的节点全着上红色，而将右边那组中的节点全着上蓝色。那么所有的边都是在红色和蓝色节点间的。$K_{3,3}$就是二分图（bipartite graph）的例子，也就是可以用两种颜色着色的图。所有这样的图都可以把它们的节点分成两组，其中同一组的成员间是没有边连接的。

最后再举个例子，图9-64中6节点图的色数为4。要知道原因，可以注意到中心的节点不能与其他任何节点颜色相同，因为它与所有节点都是连通的。因此要为它单独使用一种颜色，比方说红色。我们还至少需要两种别的颜色来为环上的节点着色。不过，如果我们试着像图9-64中所做的那样交替着色，比方着上蓝色和绿色，就会遇到问题，第五个节点的邻居既有蓝色也有绿色，因此这个例子中就需要第四种颜色——黄色。

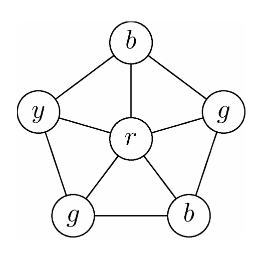


图9-64　色数为4的图

9.10.5　图着色的应用

找出一种好的图着色方案是计算机科学中另一个热门问题。例如，在第1章的全书内容简介中，我们考虑过将课程分配到时间段中，从而使学生不可能选到同一时段上课的两门课程。这样做的动机是在安排期末考试时保证学生不可能在同一时间需要参加两科考试。我们可以画一幅节点是各门课程的图，其中如果有学生同时选修了某两门课程，在表示这两门课程的节点间就存在边。

这样一来，需要多少个时段来安排考试的问题就成了计算该图的色数是多少的问题。所有颜色相同的节点都可以被安排在同一时段，因为它们之间不存在边。反过来讲，如果有一套对任何学生来说都不会引起时间冲突的安排，那么就可以把所有可安排在同一时段的课程涂成相同的颜色，因此就生成了一种颜色数量与考期时段数相同的图着色方案。

在第1章中，我们试探过基于寻找最大独立集安排考试的方法。这对寻找为图着色的好方案来说也是一种合理的试探。大家可能会期待可以为小到像图1-1中5节点图那样的图尝试所有可

能的颜色，其实这是可以的。不过，随着节点数的增加，图可能着色的种数是呈指数式增加的，而且，在找寻最少可能颜色的过程中，为那些非常大的图考虑所有可能的着色并不可行。

9.10.6　团

无向图G的团（clique）是G中满足每对节点之间都存在边的所有节点组成的集合。含k个节点的团就叫作k点团（k-clique）。图中最大团的大小就叫作该图的团数（clique number）。

✦　示例 9.31

举个简单的例子，任意完全图K_n都是由全部n个节点组成的团。事实上，对所有的$k \leq n$来说，K_n都有k点团，但如果$k > n$，则没有k点团。

图9-64中的图有大小为3的团，但没有更大的团了。这些3点团都是三角形的。因为没法再将其他节点纳入环中，所以该图中不可能有4点团。每个环节点都只连接到其他3个节点，所以4点团必定会包含环上的某个节点v、它在环上的邻居，以及中心节点。不过，v在环上的邻居之间没有边，所以没有4点团。

举个团的应用的例子，假设不像图1-1那样表示课程的冲突，而是用两个节点之间的边表示这两门课程没有学生同时选择。如此，两门有边连接的课程可以在同一时间考试。然后我们可以查找极大团（maximal clique），即不是更大的团的子集的团，而且要为同时段课程的极大团安排考试。

9.10.7　习题

(1) 对图9-4中的图而言：

  (a) 色数是多少？

  (b) 团数是多少？

  (c) 给出一个最大团的例子。

(2) 如果将(a)图9-5；(b)图9-26中的图变成无向图，那么它们的色数各是多少？可将弧当作边。

(3) 图9-5不是用平面方式表示的。该图是否为平面图？也就是说，能否重画该图，从而使该图中没有交叉的边？

(4) * 与无向图相关的3个量分别是它的度（任意节点的最大邻居数）、它的色数和它的团数。推导这3个量之间一定成立的不等关系。并解释这些关系为何一定成立。

(5) ** 设计算法，接受含n个的节点而且节点数和边数较大者为m的图，并在$O(m)$的时间内能分辨该图是否为二分图（可着2色的图）。

(6) * 我们可以把图9-64中的图一般化为具有一个中心节点以及k个在同一环上的节点，其中环上的每个节点都只与其在环上的邻居以及中心节点相连。给出该图的色数，用k的函数表示。

(7) * 对像在9.5节中讨论过的那样的无序无根树的色数有什么说法？

(8) ** 设$K_{i,j}$是取一组i个节点以及另一组j个节点，并把一组中各个节点和另一组中的每个节点用边连接后形成的图。我们看到如果$i = j = 3$，那么得到的图就不是平面图。那么对什么样的i和j来说$K_{i,j}$是平面图？

9.11　小结

图9-65中的表对我们在本章中解决的各种问题、解决这些问题的算法，以及这些算法的运行时间进行了总结。在该表中，n是图中的节点数，而m是图中节点数与弧（边）数之间的较大

者。除非另外标明，否则假设图是由邻接表表示的。

问　　题	算　　法	运行时间
最小生成树	克鲁斯卡尔算法	$O(m\log n)$
检测环路	深度优先搜索	$O(m)$
拓扑排序	深度优先搜索	$O(m)$
单一源可达性	深度优先搜索	$O(m)$
连通分支	深度优先搜索	$O(m)$
传递闭包	n次深度优先搜索	$O(mn)$
单一源最短路径	使用偏序树实现的迪杰斯特拉算法	$O(m\log n)$
	使用9.8节习题(4)实现的迪杰斯特拉算法	$O(n^2)$
所有节点对的最短路径	n次利用使用偏序树实现的迪杰斯特拉算法	$O(mn\log n)$
	n次利用使用9.8节习题(4)实现的迪杰斯特拉算法	$O(n^3)$
	利用使用邻接矩阵表示的弗洛伊德算法	$O(n^3)$

图9-65　图算法的总结

除此之外，我们还为读者介绍了图论中最关键的一些概念，包括：

- 路径和最短路径；
- 生成树；
- 深度优先搜索树和森林；
- 图着色和色数；
- 团和团数；
- 平面图。

9.12　参考文献

更多与图算法有关的材料见Aho, Hopcroft，and Ullman [1974,1983]。Hopcroft and Tarjan [1973]中首先引入了深度优先搜索来创建高效的图算法。迪杰斯特拉算法来源于Dijkstra [1959]，弗洛伊德算法来源于Floyd [1962]，克鲁斯卡尔算法来源于Kruskal [1956]，而沃夏尔算法来源于Warshall [1962]。

Berge [1962]涵盖了数学领域的图论。Lawler [1976]，Papadimitriou and Steiglitz [1982]，以及 Tarjan [1983]展示了高端的图优化技术。

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1974]. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman [1983]. *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass.

Berge, C. [1962]. The *Theory of Graphs and its Applications*, Wiley, New York.

Dijkstra, E. W. [1959]. “A note on two problems in connexion with graphs,” *Numberische Mathematik* **1**, pp. 269–271.

Floyd, R. W. [1962]. “Algorithm 97: shortest path,” *Comm. ACM* **5**:6, pp. 345.

Hopcroft, J. E., and R. E. Tarjan [1973]. “Efficient algorithms for graph manipulation,” *Comm. ACM* **16**:6, pp. 372-378.

Kruskal, J. B., Jr. [1956]. "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proc. AMS* **7**:1, pp. 48–50.

Lawler, E. [1976]. *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York.

Papadimitriou, C. H., and K. Steiglitz [1982]. *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, New Jersey.

Tarjan, R. E. [1983]. *Data Structures and Network Algorithms*, SIAM, Philadelphia.

Warshall, S. [1962]. "A theorem on Boolean matrices," *J. ACM* **9**:1, pp. 11-12.

第 10 章

模式、自动机和正则表达式

模式是具有某个可识别属性的对象组成的集合。字符串集合就是一类模式，比如C语言合法标识符的集合，其中每个标识符都是个字符串，由字母、数字和下划线组成，开头为字母或下划线。另一个例子是由只含0和1的给定大小数组构成的集合，读字符的函数可以将其解释为表示相同符号。图10-1就展示了全都可以解释为字母A的3个 7×7 数组。所有这样的数组就可以构成模式“A”。

```
0 0 0 1 0 0 0      0 0 0 0 0 0 0      0 0 0 1 0 0 0
0 0 1 1 1 0 0      0 0 1 0 0 0 0      0 0 1 0 1 0 0
0 0 1 0 1 0 0      0 0 1 1 0 0 0      0 1 1 0 1 0 0
0 1 1 0 1 1 0      0 1 0 1 0 0 0      0 1 1 1 1 1 0
0 1 1 1 1 1 0      0 1 1 1 0 0 0      1 1 0 0 0 1 1
1 1 0 0 0 1 1      1 0 0 1 1 0 0      1 0 0 0 0 0 1
1 0 0 0 0 0 1      1 0 0 0 1 0 0      0 0 0 0 0 0 0
```

图10-1　模式“A”的3个实例

与模式相关的两个基本问题是它们的定义与它们的识别，这是本章以及第11章的主题。模式的识别是诸如图10-1所示的光学字符识别（Optical Character Recognition，OCR）这样的任务中不可或缺的一部分。在某些应用中，程序中模式的识别是编译过程，也就是将程序从一种语言（比方说C语言）翻译成另一种语言（比如机器语言）的过程的一个重要部分。

模式应用在计算机科学中还有其他很多例子。模式在设计用于组成计算机和其他数字设备的电子电路的过程中扮演着关键的角色。它们也可以用在文本编辑器中，让我们可以查找特定单词或特定字符串集合的实例，比如“字母`if`之后跟着任意由`then`开头的字符序列”。大多数操作系统允许用户在命令中使用模式，例如，UNIX命令“`ls *tex`”就会列出所有以3字符序列“`tex`”结尾的名称。

人们围绕着模式的定义和识别建立起了一套庞大的知识体系。这一理论被称为“自动机理论”或“语言理论”，而其基本定义和技术都是计算机科学的核心部分。

10.1　本章主要内容

本章处理的是由字符串集合组成的模式，我们在本章中将会学习以下内容。

- “有限自动机”是一种基于图的模式指定方式。有限自动机又分为两种：确定自动机（10.2节）和非确定自动机（10.3节）。

- 可以用简单的方法把确定自动机转换成识别其模式的程序（10.2节）。
- 可以利用10.4节介绍的“子集构造”，把非确定自动机转换成识别相同模式的确定自动机。
- 正则表达式是种代数，用来描述可由自动机描述的同类模式（10.5节到10.7节）。
- 正则表达式可转换为自动机（10.8节），反之亦然（10.9节）。

我们还要在第11章中讨论串模式，其中会引入一种名为“上下文无关文法”的递归表示法来定义模式。我们将看到这种表示法可以描述没法用自动机或正则表达式表示的模式。不过，在很多情况下，文法都不如自动机或正则表达式那样容易转换为程序。

10.2　状态机和自动机

用来查找模式的程序通常有着特殊的结构。我们可以在代码中确定某些位置，在这些位置可以得知与程序寻找模式实例的过程有关的特殊信息。我们将这些位置称为状态。而程序的整体行为可以视作程序随着读入输入从一种状态转移到另一种状态。

要让这些概念变得更具体，可以考虑一个具体的模式匹配问题：“哪些英语单词按次序含有5个元音字母？”要回答这一问题，可以使用很多操作系统中都能找到的单词表。例如，在UNIX系统中可以在文件`/usr/dict/words`中找到这样的表，表中每一行都含有一个常用单词。在该文件中，一些含多个元音字母的单词是按以下次序排列的：

```
abstemious
facetious
sacrilegious
```

我们来编写一个简单的C语言程序，检查某个字符串并确定5个元音字母是否按次序出现在该字符串中。从字符串的开头开始，该程序首先会查找到`a`。我们会说该程序处于“状态0”，直到它发现一个`a`，然后它就进入“状态1”。在状态1中，它会查找字母`e`，而且当它找到一个之后，就会进入“状态2”。该程序会继续按照这种方式运行，直至到达查找字母`u`的“状态4”。如果它找到`u`，那么该单词就是按次序含有5个元音字母，这个程序就能进入一个用于接受的“状态5“。不需要再扫描单词的其余部分，因为已经可知，不管`u`后面有哪些字母，该单词都是满足条件的。

可以这样解释状态i，就是对$i = 0$、1、…、5，程序已经按次序遇到了前i个元音字母。这6个状态总结了程序在从左到右扫描其输入的过程中需要记住的所有内容。例如，在状态0中，尽管该程序在查找`a`，但它不需要记住是否已经看到了`e`。原因在于这样的`e`不可能先于任何`a`，因此不能作为序列`aeiou`中的`e`。

这种模式识别算法的核心是图10-2中的`findChar(pp,c)`函数。该函数的参数是`pp`——指向字符串的指针的地址，以及所需的字符c。也就是说，`pp`是“指向指向字符的指针的指针”。函数`findChar`会查找字符c，并且顺便会移动已给定地址的指针，直到该指针指向超过字符c或该串结尾的位置。它返回`BOOLEAN`类型的值，就是我们定义的与`int`相同的类型。正如在1.6节中讨论过的，我们预期`BOOLEAN`类型的值只有`TRUE`和`FALSE`，它们分别被定义为1和0。

在第(1)行，`findChar`会检查当前由`pp`指示的字符。如果它既不是所需的字符c，也不是C语言中标记字符串末端的字符“`\0`”，那么在第(2)行我们会移动`pp`指向的该指针。第(3)行的测试会确定我们是否因为遍历完该串而停止。如果是，就返回`FALSE`，否则前移该指针并返回`TRUE`。

```
        #include <stdio.h>

        #define TRUE 1
        #define FALSE 0
        typedef int BOOLEAN;

        BOOLEAN findChar(char **pp, char c)
        {
(1)         while (**pp != c && **pp != '\0')
(2)             (*pp)++;
(3)         if (**pp == '\0')
(4)             return FALSE;
            else {
(5)             (*pp)++;
(6)             return TRUE;
            }
        }

        BOOLEAN testWord(char *p)
        {
            /* 状态 0 */
(7)         if (findChar(&p, 'a'))
                /* 状态 1 */
(8)             if (findChar(&p, 'e'))
                    /* 状态 2 */
(9)                 if (findChar(&p, 'i'))
                        /* 状态 3 */
(10)                    if (findChar(&p, 'o'))
                            /* 状态 4 */
(11)                        if (findChar(&p, 'u'))
                                /* 状态 5 */
(12)                            return TRUE;
(13)        return FALSE;
        }

        main()
        {
(14)        printf("%d\n", testWord("abstemious"));
        }
```

图10-2 找到带有子序列aeiou的单词

在图10-2中，接下来是testWord(p)函数，它可以区分由p指向的字符串是否按次序含有所有元音字母。该函数在第(7)行前从状态0开始。在该状态中它在第(7)行调用findChar，其中第二个参数是a，用来查找字母a。如果它找到了a，findChar就会返回TRUE。因此在第(7)行如果findChar返回了TRUE，程序就会转移到状态1，其中在第(8)行会对e进行相似的测试，从第一个a之后开始扫描该字符串。因此它会继续查找元音字母，直到第(12)行，如果它找到了字母u，就会返回TRUE。如果有任何一个元音字母未被找到，控制权就会转移到第(13)行，在该行中testWord会返回FALSE。

第(14)行的主程序会测试特定的字符串“abstemious”。在实践中，我们可能会对文件中的所有单词反复使用testWord，以找出那些按次序含有5个元音字母的单词。

10.2.1 状态机的图表示

我们可以把图10-2中这种程序的行为用图表示出来，其中图的节点表示该程序的各个状态。

更重要的可能在于，可以通过设计图从而设计出程序，并机械化地将图转化成程序，要么自己动手做，要么利用某种为这个目的编写的程序设计工具。

表示程序状态的图都是有向图，它们的弧都是用字符集标记的。如果当我们在状态s时，刚好只有当看到集合C中的一个字符时才能行进到状态t，就存在从状态s到状态t的标号为字符集C的弧。这些弧叫作转换（transition）。如果x是字符集C中的某个字符，它标记了从状态s到状态t的转换，就说“进行了针对x的到状态t的转换”。在集合C为单元素集$\{x\}$这种常见的情况下，我们会使用x作为该弧的标号，而不用$\{x\}$。

我们还会给某些节点标记*接受状态*（accepting state）。当到达这些状态之一时，就找到了模式并要“接受”它。按照惯例，接受状态是用双层圆圈表示的。最后，这些节点之一会被指定为起始状态，也就是开始模式识别过程所在的状态。我们用一条不知道来自何方的进入箭头表示起始状态。这样的图就被称为*有限自动机*，或就叫*自动机*。在图10-3中可以看到自动机的一个例子。

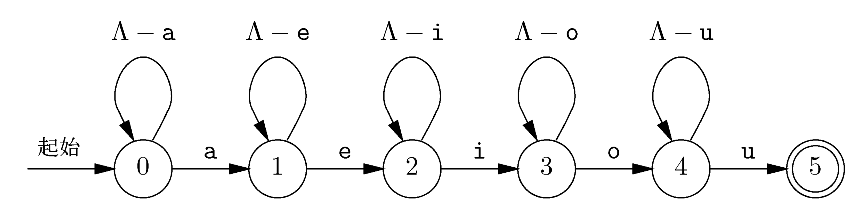


图10-3　识别含子序列aeiou的字符序列的自动机

从概念上讲，自动机的行为其实很简单。可以想象，自动机接收一列已知字符作为*输入序列*。它从起始状态开始读输入序列的第一个字符。根据第一个字符的不同，它进行的转换可能是转换到同一状态，也可能是转换到另一状态。这种转换可用自动机的图来表示。然后自动机会读第二个字符，并作出合适的转换，等等。

✦　示例 10.1

对应图10-2中testWord函数的自动机如图10-3所示。在该图中，我们使用了下面都要遵守的一个约定，用希腊字母Λ（拉姆达）代表所有大写字母和小写字母组成的集合。还要用Λ−a这样的简写形式表示除a之外所有大小写字母组成的集合。

节点0是起始状态。针对除了a之外的任意字母，我们都会保持状态0，不过遇到a就要进入状态1。同样，一旦到达状态1，就会停留在状态1，除非看到e，在看到e的情况下就要进入状态2。接下来，当看到i然后看到o时就分别到达状态3和状态4。除非看到u并进入唯一的接受状态——状态5，否则我们会停留在状态4中。再没有任何从状态5出发的转换了，因为我们不再检测待测单词的其余字符，而是要返回TRUE，声明我们已成功完成测试。

在状态0到状态4中遇到空白（或其他非字母字符）也是没有价值的，我们不会进行任何转换。在这种情况下，处理会停止，而且，因为我们现在未到达接受状态，所以会拒绝该输入。

✦　示例 10.2

接下来的例子来源于信号处理。这里不再把所有字符作为自动机可能接收到的输入，而是只允许输入0和1。我们要设计的这种特殊自动机有时也称为*反弹过滤器*（bounce filter），它接受0和1组成的序列作为输入。该自动机的目的就是“平滑”该序列，方法是将由1包围的一个0当作“噪音”，并把这个0替换为1。同样，由0包围的一个1也会被当作噪声并被0替代。

这里举一个反弹过滤器使用方式的例子，我们可以逐行扫描某数字化的黑白图像。该图像的每一行其实都是0和1组成的序列。因为图片有时候会因胶片瑕疵或拍摄问题造成有一些小点颜色错误，所以，为了减少图像中不同区域的数量，并让我们将精力放在“真实”的特色而非那些虚假的特征上，消除这样的点是很实用的。

图10-4表示的就是对应该反弹过滤器的自动机。其中的4种状态解释如下：

(a) 我们已经看到至少在一行中含两个0的一列0；

(b) 我们已经看到一列0后面跟着一个1；

(c) 我们已经看到至少有两个1的一列1；

(d) 我们已经看到一列1后面跟着一个0。

状态a被指定为起始状态，表示我们的自动机进行处理时就好像在输入之前有一个看不见的前缀0序列那样。

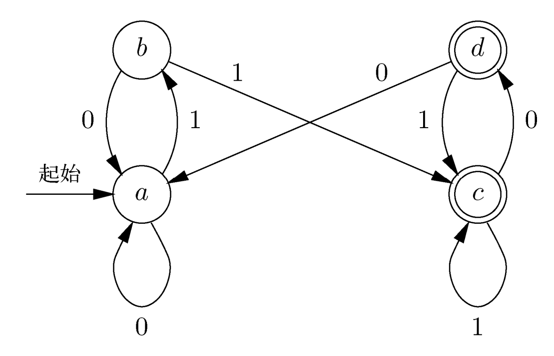


图10-4　消除虚假的0和1的自动机

接受状态是c和d。对该自动机而言，其接受过程与图10-3所示的自动机有着一些不同的含义。对图10-3所示的自动机而言，在到达接受状态时，就可以说整个输入都被接受了，包括自动机还没有读到的那些字符。①而在这里，我们想要接受状态表述“输出一个1”，还要一个表述“输出一个0”的非接受状态。在这种解释下，我们会将输入中的每一位都转化成输出中的每一位。通常输出是和输入相同的，不过有时候也会不同。例如，图10-5展示了输入为0101101时的输入、各个状态和它们的输出。

输入：		0	1	0	1	1	0	1
状态：	a	a	b	a	b	c	d	c
输出：	0	0	0	0	0	1	1	1

图10-5　图10-4中的自动机处理输入0101101时的情况模拟

我们从状态a开始，因为a是非接受状态，所以输出0。请注意，这一初始输出并不是对任意输入的回应，而是表示在初次开启设备时自动机的条件。

图10-4中从状态a出发标记了输入0的转换是到达状态a自身的。因此第二个输出还是0。第二个输入是1，而且从状态a可以进行针对1的到状态b的转换。该状态“记住了”我们已经看到过一个1，不过因为b是非接受状态，所以输出仍然是0。针对第三个输入，也就是另一个0，我

① 不过，通过为状态5加一个所有字母上的转换，我们可以修改该自动机，使其能继续读u之后的所有字母。

们又从状态b回到了状态a，而且继续发出输出0。

接下来的两个输入都是1，可以先将自动机带到状态b，然后带到状态c。对这两个1中的第一个1，我们发现自己是在状态b中，这会带来输出0。这个输出是错的，因为我们其实已经开始处理1了，但是在读完第四个输入后还不知道这一点。这种简单设计的影响在于，不管是0还是1组成的，所有的串都被右移了一位，因为在自动机意识到它已经开始处理新的串而不是“噪声”位之前，一行中已经接受了2位。在接收第5个输入时，我们就会遵循从状态b到状态c针对输入1的转换。在这一情况下，会得到第一个1输出，因为c是接受状态。

最后两个输入是0和1。0把我们从状态c带到状态d，这样我们可以记得自己已经看到了一个0。从状态d的输出依然是1，因为该状态是接受状态。最后的1将我们带回状态c并生成输出1。

自动机与其程序之间的区别

自动机是种抽象。从10.3节起将会变得明确，通过确定从起始状态到某个用相应序列标记的接受状态之间是否存在路径，自动机呈现了一种对任意输入字符序列的接受/拒绝决定。举例来说，图10-5表示的反弹过滤器自动机的行为告诉我们，该自动机拒绝ϵ、0、01、010和0101这些前缀，但它接受01011、010110和0101101这几个前缀，如图10-4所示。图10-3的自动机接受`abstemiou`这样的字符串，但拒绝`abstemious`，因为从状态5没办法到达最后的`s`。

另一方面，由自动机创建的程序能以多种方式使用这种接受/拒绝决定。例如，图10-2中的程序使用了图10-3所示的自动机，但它不是认可标记通向接受状态的路径的字符串，而是认可整行输入，也就是，接受`abstemious`而非`abstemiou`。这是绝对合理的，而且反映了我们编写程序测试按次序的5个元音字母的方式，而不管是使用了自动机或是其他的方法。据推测，只要我们到达字母`u`，该程序就会打印出整个单词而不再继续检查其余字母。

图10-4所示自动机的使用方式就更简单。我们将会看到，图10-7中对应这一反弹过滤器的程序会直接把每个接受状态转化成打印一个`1`的行动，而将每个拒绝状态转化成打印一个`0`的行动。

10.2.2　习题

(1) 设计自动机，读由0和1组成的串，并能进行下述操作。

　(a) 确定目前位置读到的序列是否有偶校验（即存在偶数个1）。特别要指出的是，如果目前为止该串有偶校验，则该自动机会接受它，而如果它具有奇校验，自动机就会拒绝它。

　(b) 检验输入串没有两个以上连续的1。也就是说，除非111是当前为止读过的输入串的子串，否则接受。

　每种状态的直觉含义各是什么？

(2) 在给定输入101001101110时，指出习题(1)中自动机的状态序列和输出。

(3) * 设计自动机，读的是单词（字符串），并分辨单词中的字母是否是已排好序的。例如，`adept`和`chilly`这样的单词中的字母就是已排好序的，而`baby`就不是，因为在第一个`b`后面有个`a`。单词一定是以空白终止的，这样自动机才会在读完所有字符后知道这一点。与示例10.1不同，这里我们必须在读完所有字符后才能接受，也就是说，必须在到达单词末端的空白之后才能接受。该自动机需要多少种状态？每种状态的直觉含义是什么？从每种状态出发的转换又有多少？总共有多少种接受状态？

(4) 设计自动机，使其能分辨字符串是否为合法的C语言标识符（字母后跟上字母、数字或下划线）后跟上空白。

(5) 编写C语言程序，实现习题(1)到习题(4)的各种自动机。

(6) 设计自动机，使其能分辨给定的字符串是否为第三人称单数代词（he、his、him、she、her或hers）后跟上空白。

(7) * 将习题(6)设计的自动机转换成C语言函数，并在程序中使用该函数找到某给定字符串中所有出现第三人称单数代词子串的位置。

10.3　确定自动机和非确定自动机

使用自动机进行的最基本的操作之一是接受一系列的符号 $a_1a_2\cdots a_k$，并从起始状态起循着一条由标号依次为这些符号的弧组成的路径行进。也就是说，对i = 1、2、…、k来说，a_i 都是集合S_i中作为路径上第i条弧标号的成员。构建这一路径及其状态序列的过程就是自动机对输入序列 $a_1a_2\cdots a_k$ 的模拟（simulating）。可以说这一路径标号为 $a_1a_2\cdots a_k$，当然，它也可能有其他标号，因为给路径上的弧提供标号的各集合S_i可能各自含有很多字符。

✦　示例 10.3

我们在图10-5中进行过一次这样的模拟，其中模仿了图10-4中的自动机对序列0101101的处理。另外，以图10-3中用来识别单词中是否含有序列aeiou的自动机为例，考虑对字符串adept的处理。

我们从状态0中开始。从状态0出发的转换有两次，一次是针对字符集Λ－a的转换，另一次是针对单独一个字母a的。因为adept的第一个字符就是a，所以要遵循后一个转换，这把我们带到了状态1。从状态1出发，又有针对Λ－e和e的转换。因为第二个字符是d，所以必须遵循前一种转换，因为Λ－e包含除了e之外的所有字母。这把我们再次留在状态1中。因为第三个字母是e，所以要循着从状态1出发的第二种转换，将我们带到状态2。adept的最后两个字母都在集合Λ－i中，所以下两次转换都是从状态2到状态2。因此在状态2中就完成了对adept的处理。相应的状态转换序列如图10-6所示。因为状态2不是接受状态，所以我们没有接受输入adept。

输入：		a		d		e		p		t	
状态：	0		1		1		2		2		2

图10-6　对10-3中的自动机针对输入adept的模拟

有关自动机输入的术语

在这里将要讨论的例子中，自动机的输入是字符，比如字母和数字，而且将输入当作字符并将输入序列当作字符串是很方便的。我们在这里一般会使用这一术语，不过偶尔会将“字符串”简称为“串”。不过，在有些应用中，自动机要转换的输入是从比ASCII字符集更广泛的集合中选出的。例如，编译器可能会把while这样的关键词看作单个输入符号，我们将这种情况用加粗的字符串while表示。因此有时候我们会把这种单独的输入称作“符号”而非“字符”。

10.3.1　确定自动机

在10.2节中讨论过的自动机有个重要的属性。对任意状态*s*和任意输入字符*x*来说，至多只有一种从状态*s*出发的转换的标号中含有*x*。这样的自动机就称为*确定自动机*。

为给定输入序列模拟确定自动机是很简单的。在任意状态*s*中，给定下一个输入字符*x*，考虑从*s*出发的每种转换的标号。如果我们找到标号含*x*的转换，那么该转换就指向适当的下一个状态。如果没有含*x*的转换，那么该自动机就“死机”了，而且不能再继续处理输入，就像图10-3中的自动机在到达状态5后就会停机那样，因为它知道自己已经找到了子序列`aeiou`。

将确定自动机转变为程序是很容易的。我们为每个状态编写一段代码。对应状态*s*的代码会检查它的输入，并决定应该遵循从*s*出发的哪种转换（如果存在这样的转换）。如果选定了从状态*s*到状态*t*的转换，那么必须安排表示状态*t*的代码接着表示状态*s*的代码执行，可能是通过`goto`语句来实现。

✦　示例 10.4

这里我们编写了一个对应图10-4所示反弹过滤器自动机的函数`bounce()`。变量`x`是用来从输入中读字符的。状态*a*、*b*、*c*和*d*将分别用标号`a`、`b`、`c`和`d`来表示，而且要使用标号`finis`表示程序的结尾，也就是在输入中遇到0和1之外的字符时会到达的地方。

代码如图10-7所示。例如，在状态*a*中我们会打印字符0，因为*a*是非接受状态。如果输入字符是0，就停留在状态*a*，而且如果输入字符是1，就进入状态*b*。

```
void bounce()
{

        char x;

        /* 状态 a */
a:      putchar('0');
        x = getchar();
        if (x =='0')  goto a; /* transition to state a */
        if (x =='1')  goto b; /* transition to state b */
        goto finis;

        /* 状态 b */
b:      putchar('0');
        x = getchar();
        if (x == '0') goto a; /* transition to state a */
        if (x == '1') goto c; /* transition to state c */
        goto finis;

        /* 状态 c */
c:      putchar('1');
        x = getchar();
        if (x == '0') goto d; /* transition to state d */
        if (x == '1') goto c; /* transition to state c */
        goto finis;

        /* 状态 d */
d:      putchar('1');
        x = getchar();
        if (x =='0')  goto a; /* transition to state a */
        if (x =='1')  goto c; /* transition to state c */
        goto finis;

finis:  ;

}
```

图10-7　实现图10-4中确定自动机的函数

在“自动机”的定义中没有要求从某给定状态出发的转换的标号必须是不相交的，如果集合没有相同的成员则说它们是不相交的，即它们的交集为空。如果有图10-8所示的这种图，其中针对输入x有从状态s到状态t和状态u的转换，这样一来该自动机要如何用程序来实现就不是很清楚了。也就是说，在执行对应状态s的代码时，如果发现x是下一个输入字符，就得知接下来一定要进入表示状态t的代码的开头，而且还要进入表示状态u的代码的开头。因为程序一次不能到达两个位置，所以要如何模拟从状态出发的转换具有相同标号的自动机是很不明朗的。

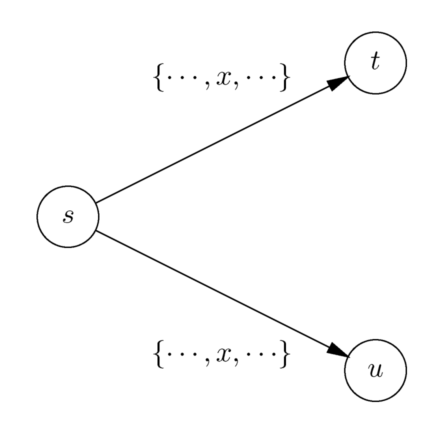


图10-8 从状态s出发的针对输入x的非确定转换

10.3.2 非确定自动机

非确定自动机可以具有从某一状态出发的包含相同符号的两个或多个转换，但这不是必须的。请注意，严格地讲，确定自动机也是一种非确定自动机，它只是刚好没有针对同一符号的多种转换。一般来说“自动机”都是不确定的，不过我们在强调自动机不是确定自动机时还是会使用“非确定自动机”的说法。

正如上文提过的，非确定自动机不能直接用程序实现，不过它们对这里将要讨论的若干应用来说是很实用的概念工具。此外，通过利用10.4节中将要介绍的“子集构造”，可以将任意非确定自动机转换成接受相同字符串集合的确定自动机。

10.3.3 非确定自动机的接受

在我们试图模拟针对输入字符串$a_1a_2\cdots a_k$的非确定自动机时，可能发现同一个字符是多条路径的标号。习惯上讲，如果至少有一条由某输入编辑的路径可以通向接受状态，就可以说非确定自动机接受这一输入字符串。以接受状态结尾的那一条路径，要比任意数量以非接受状态结尾的路径更重要。

不确定性和猜测

认为不确定性让自动机可以“猜测”是种看待不确定性的实用方式。如果我们不知道在某给定状态中要对某给定的输入字符做什么，就可以对下一个状态做出若干选择。因为由带向接

受状态的字符串标记的任意路径会被解释为接受，所以非确定自动机其实被赋予了进行一次正确猜测的信用，而不管它还会造成多少次错误猜测。

✦ 示例 10.5

反性别歧视言论联盟（League Against Sexist Speech，LASS）希望找到含单词man的性别歧视文字。他们不止想捕获ombudsman（特派员）这样的构词，还希望捕获诸如maniac（狂人）或emancipate（解放）这样形式更为微妙的歧视。LASS计划设计一个使用自动机的程序，该程序会扫描字符串，并会在它从输入中任意位置找到字符串man时“接受”该输入。

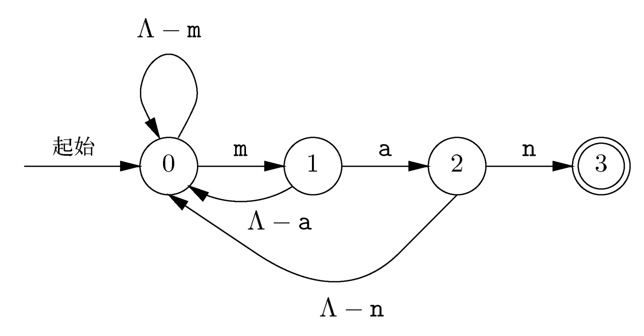


图10-9　可识别大多数（而非全部）以man结尾的字符串的确定自动机

大家可能首先会尝试如图10-9所示的确定自动机。在该自动机中，状态0，也就是起始状态，表示的是我们还没看到man这几个字母时的情况。状态1是用来表示我们已经看到m的情形，在状态2中我们已经识别了ma，而在状态3中我们已经看到了man。在状态0、状态1和状态2中，如果我们没有看到想找的字母，就回到状态0并再次尝试。

不过，图10-9并不能很正常地完成处理。在处理command这样的输入时，当它读c和o时会停留在状态0中。在读第一个m时它会进入状态1，不过第二个m又会把它带回状态0，随后它就无法离开状态0了。

可以正确识别内嵌了man的字符串的非确定自动机如图10-10所示。关键的革新在于，我们在状态0中会猜测m是否标志着man的开始。因为该自动机是非确定自动机，它允许同时猜测“是”（由从状态0到状态1的转换表示）和“否”（由可以对包括m在内的所有字母执行从状态0到状态0的转换这一事实表示）。因为非确定自动机的接受需要的不过是一条通向接受状态的路径，所以我们可以受益于这两种猜测。

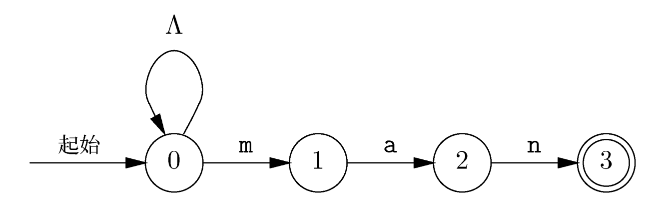


图10-10　可识别所有以man结尾的字符串的非确定自动机

图10-11展示了图10-10中的非确定自动机在处理输入字符串command时的行动。在回应c和o时，该自动机只能停留在状态0中。在输入第一个m时，自动机可以选择进入状态0或状态1，

因此它同时进入了这两个状态。在处理第二个m时，从状态1是没办法继续行进的，所以该分支就成了一条“死路”。不过，从状态0可以再次进入状态0或状态1，这里又同时进入这两种状态。当输入a时，可以从状态0到达状态0，并从状态1到达状态2。同样，在输入n时，可以从状态0到达状态0，并且从状态2到达状态3。

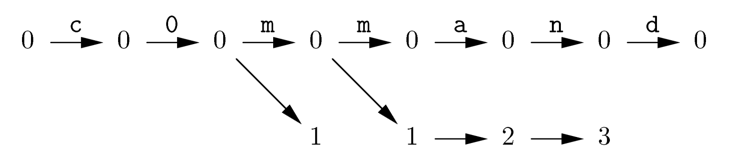


图10-11　模拟图10-10中的非确定自动机处理输入串command的情况

因为状态3是接受状态，所以在该点处我们可以接受这一输入。[①]对接受状态而言，在看到comman后也处在状态0中这一事实是无关紧要的。最后的转化是针对输入d的，从状态0到状态0。请注意状态3不会针对任意输入行进到任何位置，所以该分支也完结了。

还要注意，图10-9中展示的用来处理未接收到单词man后一个字符这种情况的回到状态0的转换，在图10-10中是不必要的，因为在图10-10中我们看到输入man时不一定要沿着序列从状态0到状态1再到状态2最后到状态3。因此，虽然状态3看起来“已死”，而且在看到man时已终止计算，但是我们在看到man时也停留在状态0中。该状态允许我们在处理manoman这样的输入时，于读第一个man期间停留在状态1中，并在读第二个man时行经状态1、状态2和状态3，以此来接受manoman这样的输入。

当然，图10-10的设计尽管很动人，但不能直接转换为程序。我们将在10.4节中看到如何把图10-10转换成只含4个状态的确定自动机。与图10-9不同的是，该确定自动机可以正确地识别所有出现man的单词。

尽管可以把任意非确定自动机转换成确定自动机，但并非总是像图10-10所示的情况这般幸运。在图10-10中的情况下，可以看到对应的确定自动机的状态不会多于原非确定自动机的状态，也就是各有4个状态。但事实上，还存在另外一些非确定自动机，与它们对应的确定自动机会含有更多状态。一个含n种状态的非确定自动机有可能只能转换成含2^n个状态的确定自动机。下一个示例正好就是确定自动机的状态要比非确定自动机的状态多得多的情况。因此，对同一个问题而言，设计非确定自动机可能比设计确定自动机简单得多。

✦ 示例 10.6

当本书作者之一Jeffrey D.Ullman 之子Peter Ullman上四年级时，他的一位老师试图通过为学生们布置一些“部分换位构词”问题来增加他们的词汇量。该老师每周会给学生们布置一个单词，并要求他们找出使用该单词的一个或多个字母可以构成的所有单词。

有那么一周，该老师布置的单词是Washington，本书的两位作者聚在一起，决定进行一次穷举查找，看看到底可能形成多少个单词。利用/usr/dict/words文件与含3个步骤的过程，我们找到了269个单词，其中有以下5个含7个字母的单词：

① 请注意，图10-10中的自动机就像图10-3中的自动机那样，在看到它查找的模式时就会接受，而不是在单词的结尾接受。当我们最终把图10-10转换成确定自动机时，就可以根据它设计能打印整个单词的程序了，就像图10-2中的程序那样。

```
agonist
goatish
showing
washing
wasting
```

因为字母的大小写对本问题来说不重要，所以第一步就是要把词典中所有的大写字母全部转化为小写字母。执行这一任务的程序是很简单的。

第二步是选取只含来自集合S={a，g，h，i，n，o，s，t，w}中的字母（washington中的字母）的单词。图10-12中的确定自动机就能完成该任务。newline字符是/usr/dict/words中标记行尾的字符。如果我们遇到newline之外的其他任意字符，就不用进行转换，而且自动机决不会到达接受状态1。如果在只读到washington中的字母后遇到newline，就进行从状态0到状态1的转换并接受该输入。

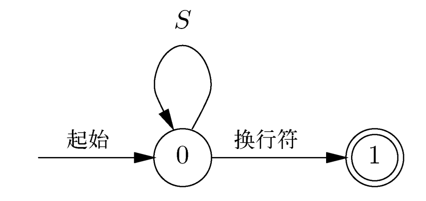


图10-12　检测由washington中出现的字母所构成单词的自动机

图10-12中的自动机接受hash这样的单词，也就是相应字母出现的次数多于washington本身中字母出现次数的单词。因此，我们的第三步也是最后一步就是，排除那些包含3个或更多n，或是包含两个或更多S中其他字符的单词。这一任务也可以由自动机来完成。例如，图10-13中的自动机接受的是至少有两个a的单词。我们会停留在状态0中，直至看到a，在这种情况下就进入状态1。接着会保持状态1，直到看到第二个a，才进入状态2并接受该输入。该自动机接受那些因为有太多a而不能用washington部分换位构词得到的单词。在这种情况下，我们想要的刚好是那些在处理过程中从不会让自动机进入接受状态2的单词。

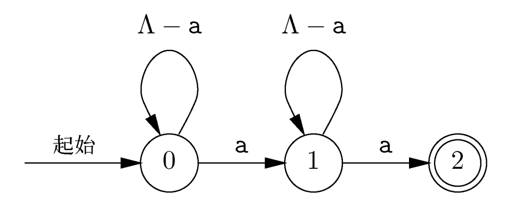


图10-13　如果输入存在两个a就接受该输入的自动机

图10-13所示的自动机是确定自动机。不过，它只表示了某单词可被图10-12中的自动机接受却仍不是washington经部分换位构词得到的单词的9种原因之一。要接受具有washington中某个字母太多实例的全部单词，我们可以使用图10-14中的非确定自动机。

图10-14从状态0中开始，而且它针对任意字母的一种选择就是留在状态0中。如果输入字符是washington中的任意一个字母，就有另一种选择；该自动机还会猜测它应该转换到这样一个状态，该状态的功能是记住该字母已经出现过一次。例如，针对字母i，我们有进入状态7的选择。然后我们就会留在状态7中，直到看到另一个i，从而进入作为接受状态之一的状态8。回想一下，在该自动机中，接受就意味着输入字符串不是由washington经过部分换位构词得到的单词，在这里描述的情况中就是因为该单词含有两个i。

因为在washington中有两个n，所以对n的处理有些不同。自动机在看到一个n后会进入状态9，而在看到第二个n后会进入状态10，接着在看到第三个n时才进入接受状态11。

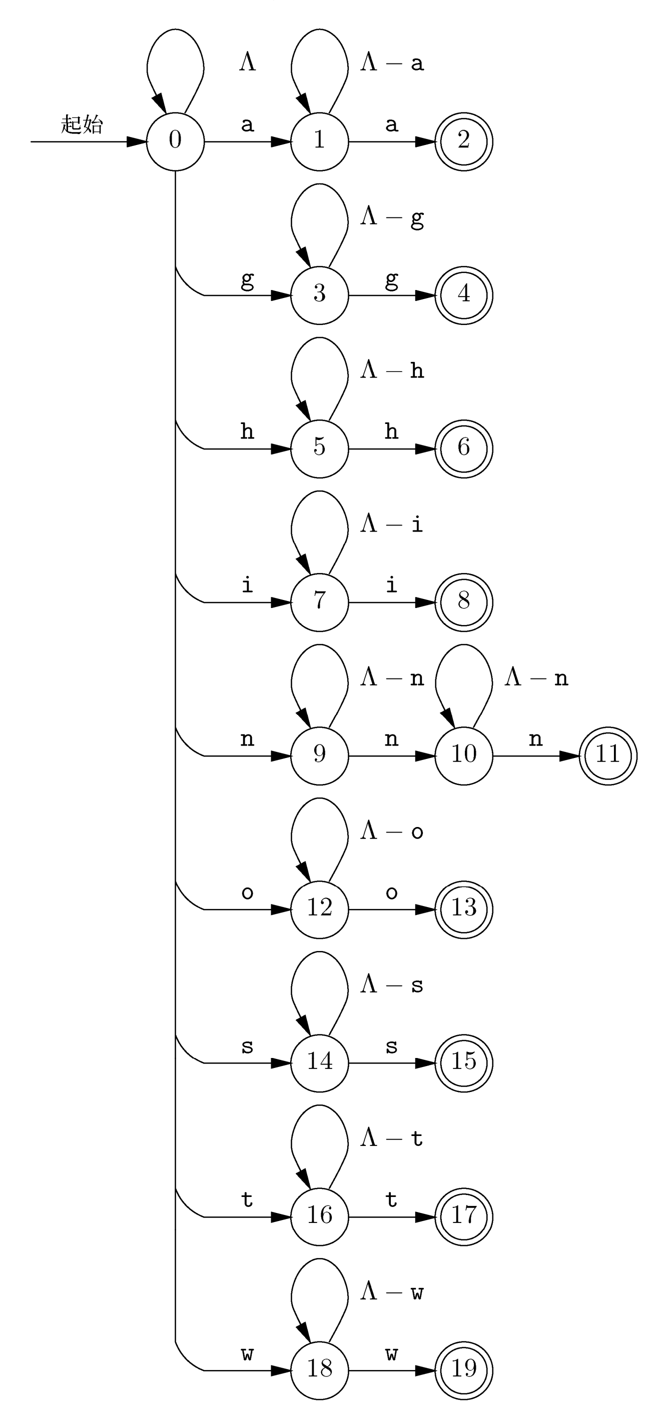


图10-14 检测含有一个以上的a、g、h、i、o、s、t或w，或者两个以上n的单词的非确定自动机

例如，图10-15展示了读输入字符串shining之后的所有状态。因为我们在读第二个i后会进入接受状态8，所以shining不是由washington经过部分换位构词得到的单词，即便它因为只含有washington中可以找到的字母而能被图10-12中的自动机接受。

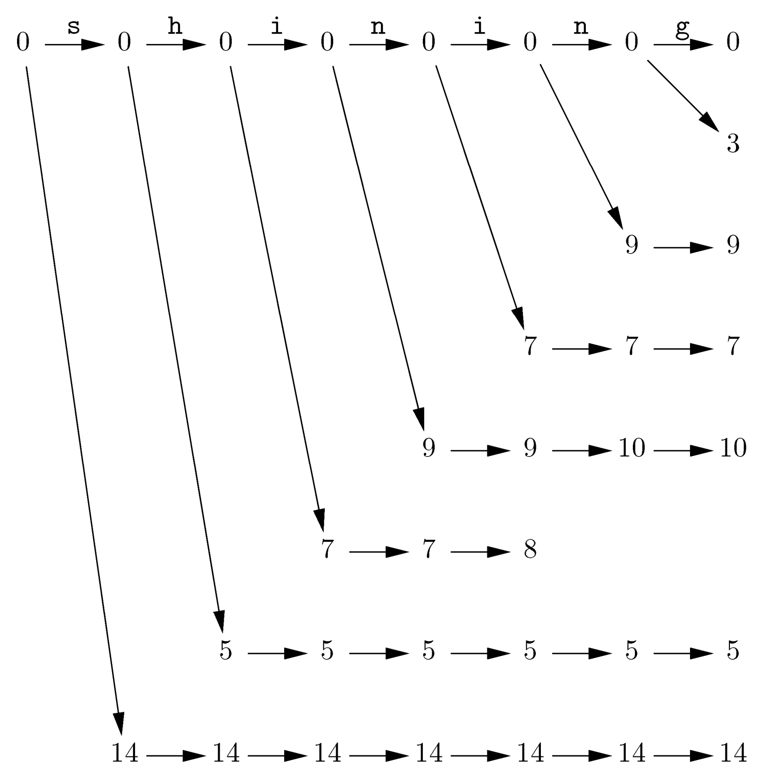


图10-15　图10-14中的非确定自动机处理输入字符串shining时进入的状态

总结下来，我们的算法由以下3步组成。

(1) 首先将词典中的所有大写字母转换成小写字母。

(2) 找到10-12中的自动机接受的所有单词，这些单词只由washington中的字母组成。

(3) 从步骤(2)得到的单词中删除图10-14中非确定自动机接受的所有单词。

该算法是在/usr/dict/words文件中找到可由Washington经过部分换位构词得到的单词的简单方法。当然，必须找到某种合理的方式来模拟图10-14中的非确定自动机，我们将在10.4中讨论如何完成这一任务。

10.3.4　习题

(1) 编写C语言程序，实现图10-9中确定自动机的算法。

(2) 设计确定自动机，使其能正确地找出字符串中出现的所有子字符串man。并将该自动机实现为程序。

(3) LASS希望检测出所有含字符串man、son和father的单词。设计非确定自动机，使其只要找到这3个字符串中任意一个就接受相应的输入字符串。

(4) * 设计确定自动机，使其可以解决习题(3)中的问题。

(5) 模拟图10-9和图10-10中的自动机处理字符串summand时的情况。

(6) 模拟图10-14的自动机处理以下字符串的情况。

(a) `saint`
(b) `antagonist`
(c) `hashish`

其中哪些字符串会被接受？

(7) 可以用具有状态、输入和接下来这些属性的关系来表示自动机。这样做的目的是，如果(s，x，t)是个元组，那么输入符号x就是从状态s到状态t的转换的标号。如果该自动机是确定自动机，那么该关系合适的键是什么？如果该自动机是非确定自动机呢？

(8) 如果只是想在给定某状态和某输入符号的情况下找出接下来的（一个或一些）状态，大家会建议用什么数据结构来表示习题(7)中的关系？

(9) 将如下图所示的自动机表示为关系。

(a) 图10-10
(b) 图10-9
(c) 图10-14

可以使用椭圆来表示 $\Lambda - m$ 这样针对含大量字母的集合的转换。

不编程找到部分换位构词形成的单词

顺便提一句，我们可以使用UNIX系统的命令，几乎不进行编程就实现示例10.6中的3步算法。对步骤(1)，可以使用UNIX命令

```
tr A-Z a-z </usr/dict/words
```
(10.1)

把大写字母转化成小写字母。对步骤(2)，可以使用命令

```
egrep '^[aghinostw]*$'
```
(10.2)

粗略地讲，就是定义了图10-12中那样的自动机。对步骤(3)，可以使用命令

```
egrep -v 'a.*a|g.*g|h.*h|i.*i|n.*n.*n|o.*o|s.*s|t.*t|w.*w'
```
(10.3)

该命令指定了类似图10-14中自动机的事物。整个任务可以使用以下三元素管道来完成：

```
(10.1) |(10.2) |(10.3)
```

也就是说，整个命令是通过用表示各行的文本替换各行形成的。竖线，或者说“管道”符号，使得左侧命令的输出可以成为右侧命令的输入。我们将在10.6节中讨论`egrep`命令。

10.4　从不确定到确定

在本节中我们将会看到，每一个非确定自动机都可以被确定自动机替代。正如我们已经看到的，在执行某些任务时，考虑非确定自动机有时要更简单些。不过，因为根据不确定自动机编写程序不如根据确定自动机编程那样容易，所以找到一种将不确定自动机变形为等价的确定自动机的算法是很重要的。

10.4.1　自动机的等价性

在10.3节中，我们已经看到两种接受观。在某些示例中，比如在示例10.1（含有子序列`aeiou`

的单词）中，接受就意味着整个单词被接受，即便我们可能没有扫描完整个单词。而在另一些例子中，比方说示例10.2的反弹过滤器中，或是图10-12所示的自动机（字母全在washington中的单词）中，只有在我们想对从启动自动机以来已经看到的确切输入表示认可时才接受该输入。因此，在示例10.2中，我们接受所有能带来输出1的输入序列。在图10-12中，只有在已经看到newline字符，知道已经看到整个单词时才接受该输入。

当谈论正式的自动机行为时，我们只需要第二种解释（当前的输入被接受）。严格地讲，假设*A*和*B*是两个自动机（确定或不确定）。如果*A*和*B*接受相同的输入字符串集合，就说它们是等价的。换句话说，如果 $a_1a_2\cdots a_k$ 是任意符号串，那么以下两个条件是成立的。

(1) 如果从*A*的起始状态到*A*的某个接受状态存在以 $a_1a_2\cdots a_k$ 标记的路径，那么从*B*的起始状态到*B*的某个接受状态也存在以 $a_1a_2\cdots a_k$ 标记的路径。

(2) 如果从*B*的起始状态到*B*的某个接受状态存在以 $a_1a_2\cdots a_k$ 标记的路径，那么从*A*的起始状态到*A*的某个接受状态也存在以 $a_1a_2\cdots a_k$ 标记的路径。

✦　示例 10.7

考虑图10-9和图10-10中的自动机。正如我们在图10-11中注意到的，图10-10中的自动机接受输入字符串comman，因为该字符序列在图10-10中标记了路径 $0\to0\to0\to0\to1\to2\to3$，而且这一路径是从起始状态出发，到达了一个接受状态。不过，在图10-9所示的确定自动机中，可以验证由comman标记的路径只有 $0\to0\to0\to1\to0\to0\to0$。因此如果图10-9是自动机*A*，而图10-10是自动机*B*，就违背了上述第(2)点，这样就表明这两个自动机不是等价的。

10.4.2　子集构造

我们现在将会看到，如何通过构造等价的确定自动机来"消除自动机的不确定性"。这一技巧叫作子集构造，而且它的本质就如图10-11和图10-15所示，在这两幅图中我们模拟了处理特殊输入的非确定自动机。从这两幅图中我们注意到，在任何给定的时间，非确定自动机都在某一状态集合中，而且这些状态都出现在模拟图的同一列中。也就是说，在读了某输入列 $a_1a_2\cdots a_k$ 之后，非确定自动机就"在"那些从起始状态出发沿着标记有 $a_1a_2\cdots a_k$ 的路径可以到达的状态中。

✦　示例 10.8

在读完输入字符串shin之后，图10-15所示的自动机处在状态集合{0，5，7，9，14}中。这些状态都出现在第一个n后的一列中。在读下一个i后，它处在状态集合{0，5，7，8，9，14}中，而在读了接下来的n后，在状态集合{0，5，7，9，10，14}中。

现在就有了如何把非确定自动机*N*转换为确定自动机*D*的线索。*D*的状态各自是*N*的状态的集合，而且*D*中状态间的转换是由*N*的转换确定的。要看到如何构建*D*的转换，设*S*是*D*的某个状态，而且*x*是某输入符号。因为*S*是*D*的状态，所以它是由*N*的状态组成的。定义集合*T*是自动机*N*中那些状态*t*，这些状态满足存在*S*中的状态*s*，以及自动机*N*针对包含输入符号*x*的集合的从*s*到*t*的转换。那么在自动机*D*中我们就放置一个在针对符号*x*的从*S*到*T*的转换。

示例10.8展示了多个针对输入符号的从一个确定状态到另一个确定状态的转换。在当前的确定状态是{0，5，7，9，14}，而且输入符号是字母i时，我们在该示例中看到，根据图10-14中的非确定自动机，接下来的不确定状态集是*T* = {0，5，7，8，9，14}。由针对输入符号*n*的

这一确定状态可知，接下来的不确定状态集是U = {0，5，7，9，10，14}。这两个确定转换如图10-16所描述的那样。

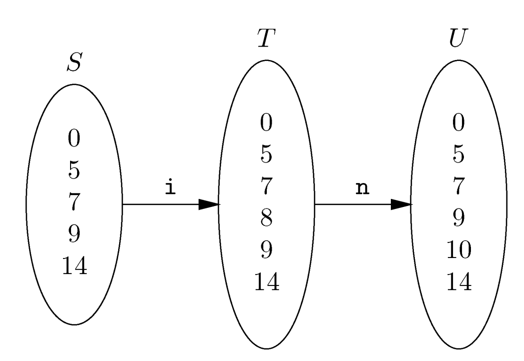


图10-16　确定状态S、T和U之间的转换

现在我们知道该如何在确定自动机D的两个状态之间构建转换了，不过需要确定自动机D确切的状态集、D的起始状态，以及D的接受状态。我们要用归纳法来构建D的状态。

依据。如果非确定自动机N的起始状态是s_0，那么确定自动机D的起始状态是$\{s_0\}$，也就是只含s_0这一个元素的集合。

归纳。假设已经确定了N的状态集S是D的一个状态。依次考虑每个可能的输入字符x。对某个给定的x，设T是N的状态t构成的集合，其中状态t满足对S中的某个状态s而言，存在标号含x的从s到t的转换。那么集合T就是D的一个状态，而且存在针对输入x的从S到T的转换。

D的接受状态是N的状态集中至少包含N的一个接受状态的。这从直觉上讲是说得通的。如果S是D的状态而且是N的状态集，那么能把D从其起始状态带到状态S的输入 $a_1a_2\cdots a_k$ 也能把N从其起始状态带到S中的所有状态。如果S含有某个接受状态，那么 $a_1a_2\cdots a_k$ 会被N接受，而且D也一定会接受该输入。因为D在接收输入 $a_1a_2\cdots a_k$ 时只会进入状态S，所以S肯定是D的接受状态。

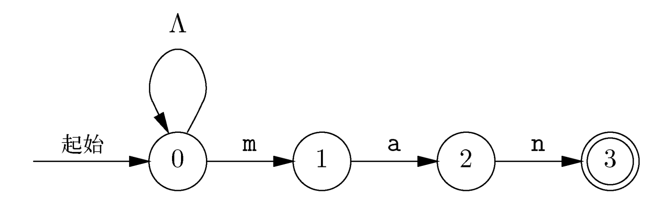


图10-17　识别以man结尾字符串的非确定自动机

✦ 示例 10.9

图10-17重现了图10-10所示的非确定自动机，我们来把它转换成确定自动机D。先从D的起始状态{0}开始。

这一构建过程的归纳部分要求我们查看D的每个状态，并确定它的转换。对{0}而言，只需要询问状态0通向哪里。分析图10-17得到的答案是，对除了m之外的任意字母，状态0只能进入状态0，而对输入m，它同时通向状态0和状态1。因此自动机D需要已经具备的状态{0}和我们必须添加的状态{0，1}。目前为止已经为D构建的转换和状态如图10-18所示。

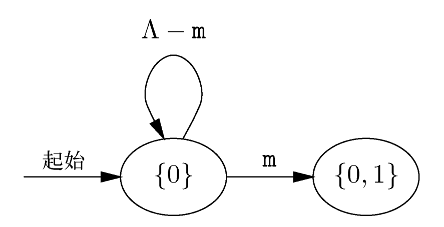


图10-18　状态{0}及其转换

接下来，必须考虑从状态{0，1}出发的转换。再次研究图10-17，我们看到，对除了m和a之外的所有输入，状态0只通向状态0，而状态1则不能通向任何地方。因此，存在从状态{0，1}到状态{0}的转换，标记为除了m和a之外的所有字母。对输入m，状态1还是不能通向任何地方，不过状态0会同时通向状态0和状态1。因此，存在从状态{0，1}到其本身的转换，标记为m。最后，对输入a，状态0只会通向它自己，不过状态1是通向状态2的。因此存在标记为a的从状态{0，1}到状态{0，2}的转换。目前为止*D*已经构建起的部分如图10-19所示。

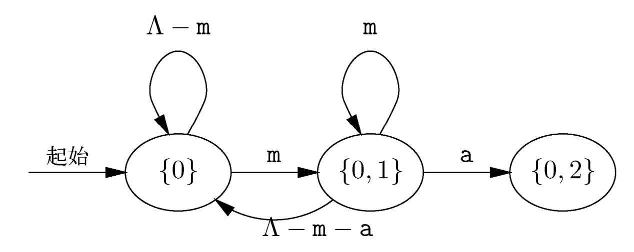


图10-19　状态{0}和{0，1}，以及它们的转换

现在需要构建从状态{0，2}出发的转换。对除m和n之外的所有输入，状态0只能通向状态0，而状态2则哪里都去不了，因此存在从{0，2}到{0}的转换，标记为除了m和n之外的所有字母。对输入m，状态2不通向任何状态，而状态0则同时通向状态0和状态1，因此标记为m的从状态{0，2}到状态{0，1}的转换。对输入n，状态0只通向它本身，而状态2会通向状态3。因此存在标记为n的从状态{0，2}到状态{0，3}的转换。*D*的该状态是接受状态，因为它包含了图10-17中的接受状态——状态3。

最后，必须给出从状态{0，3}出发的转换。因为对任何输入，状态3都不通向任何状态，从状态{0，3}出发的转换只反映从状态0出发的转换，因此会与{0}通向相同的状态。因为从状态{0，3}出发的转换不会将我们带到尚未见过的*D*的状态，所以对*D*的构造就完成了。完整的确定自动机如图10-20所示。

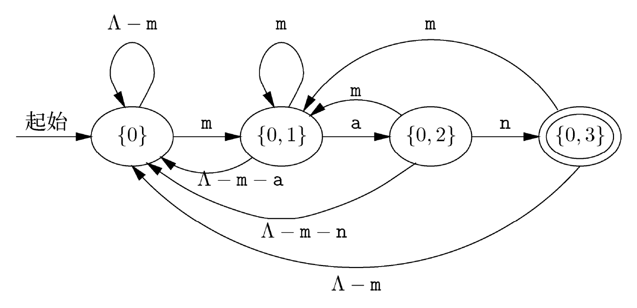


图10-20　确定自动机*D*

请注意，这一确定自动机可以正确地接受所有以man结尾的字母串，而且只接受以man结尾的字母串。直觉上讲，只要字符串目前为止不是以m、ma、man结尾，该自动机就会处在状态{0}中。状态{0，1}意味着目前为止看到的字符串是以m结尾的，状态{0，2}表示当前的字符串是以ma结尾，而状态{0，3}则说明当前字符串的结尾为man。

✦ 示例 10.10

图10-17中的非确定自动机有4个状态，而图10-20中与它等价的确定自动机也有4个状态。如果所有的非确定自动机都能转换成小型确定自动机就好了，例如，在编译编程语言中常用到的某些自动机其实可以转换成相当小的确定自动机。不过，不能保证确定自动机一定很小，具有k个状态的非确定自动机有可能最终被转换成状态多达2^k个的确定自动机。也就是说，对非确定自动机状态集的幂集中的每个成员，确定自动机都有相应的状态。

举个会得到很多状态的例子，考虑一下10.3节图10-14中的自动机。因为该非确定自动机有20种状态，可以想象一下，通过子集构造构建的确定自动机可能有约2^{20}个，或者说是超过100万个状态，这些状态全都是{0, 1, …, 19}的幂集的成员。实际结果并没有这么糟，但也存在相当多的状态。

我们不会尝试画出与图10-14中非确定自动机等价的确定自动机。不过，可以考虑一下实际上需要哪些状态集。首先，因为对每个字母都有从状态0到其自身的转换，所以实际看到的所有状态集都包含0。如果字母a尚未输入，就不能到达状态1。不过，如果刚好已经看到一个a，不管还看到些什么，我们都将在状态1中。我们还可以为washington中其他任何字母得出相似的论断。

如果在状态0中开始图10-14，并为其提供属于washington中所出现字母的子集的一列字母，然后除了在状态0中之外，还可以在状态1、3、5、7、9、12、14、16和18的某个子集中。通过恰当地选择输入字母，我们可以安排在这些状态集的任意一个中。因为有 $2^9=512$ 个这样的集合，所以在与图10-14等价的确定自动机中至少有这么多个状态。

不过，其中的状态要更多，因为字母n在图10-14中得到了特殊处理。如果在状态9中，我们还可以在状态10中，而且如果已经看到两个n，其实就将同处状态9和状态10中。因此，尽管对其他8个字母来说都只有两个选择（比如，对字母a，要包含状态1，要么不含状态1），而对字母n来说，共有3种选择（既不含状态9也不含状态10、只包含状态9，或者同时包含状态9和状态10）。因为至少存在 $3\times2^8=768$ 种状态。

不过这并非全部。如果到目前为止的输入以washington中的字母之一结束，而且我们之前已经看到足够多的该字母，那么我们也应该在对应该字母的接受状态中，比方说，对a来说就是状态2。然而，我们在处理相同输入后不可能在两个接受状态中。为增加的状态集计数变得更麻烦了。

假设接受状态2是该集合的成员。那么可知1也是该集合的成员，0当然也是，不过我们仍然对与除a之外的字母对应的状态拥有所有选择，这类集合的数量是 3×2^7，或者说是384。如果我们的集合包含接受状态4、6、8、13、15、17或19，这样的道理也同样实用，在每种情况中都有384个集合包含相应的接受状态。唯一的例外就是含接受状态11（就是状态9和状态10也出现）的情况。在该情况下，只有 2^8=256 种选择。因此，该等价确定自动机的状态总数为：

$$768+8\times384+256=4864$$

第一项768表示那些不含接受状态的集合的数量。接下来的一项，表示的是分别包含与除n之外其他8个字母对应的接受状态的8类集合的数量，第三项256则表示含状态11的集合的数量。

关于子集构造的想法

子集构造是相当不好理解的。特别是确定自动机的状态可以是非确定自动机的状态集这一思路可能需要大家耐心思考才能想通。不过，这种把结构化对象（状态集）与原子对象（确定自动机的各个状态）融合成同一对象的概念，是计算机科学中的一种重要概念。我们已经在程序中看到过这一概念，而且经常必须用到它。例如，函数的参数，比方说L，表面上看是原子的，而对其进行更仔细的检查可能发现它是有着复杂结构的对象，比如，具有连接到其他记录的字段从而能形成表的记录。同样，图10-20中确定自动机D的状态{0，2}也可以用简单的名称“5”或“a”来代替。

10.4.3　子集构造起效的原因

很明显，如果D是利用子集构造从非确定自动机N构建的，那么D就是确定自动机。原因在于，对每个输入符号x和D的每个状态S而言，我们定义了D的某特定状态T，它满足从S到T的转换的标号中包含x。不过如何得知自动机N和D是等价的呢？也就是说，我们需要知道，对任意输入序列$a_1a_2\cdots a_k$，当且仅当N接受$a_1a_2\cdots a_k$时，在下列情况下，自动机D到达的状态S是种接受状态。

(1) 从起始状态开始；

(2) 并且沿着标记为$a_1a_2\cdots a_k$的路径行进。

请记住，当且仅当从N的起始状态有一条标记为$a_1a_2\cdots a_k$的路径能到达N的某个接受状态时，N才会接受$a_1a_2\cdots a_k$。

D所做的事情与N所做的事情之间的联系就更紧密了。如果D具有从它的起始状态到标记为$a_1a_2\cdots a_k$的状态的路径，那么被视为自动机N的某个状态集的集合S，就刚好是从N的起始状态开始沿着某标记为$a_1a_2\cdots a_k$的路径所能到达的状态组成的集合。这种关系如图10-21所示。因为我们已经定义了，只有在S中的某一成员是N的接受状态时，S才是D的接受状态，所以要得出D和N都接受$a_1a_2\cdots a_k$或都不接受$a_1a_2\cdots a_k$，也就是要得出D和N是等价的，就只需要图10-21所示的关系。

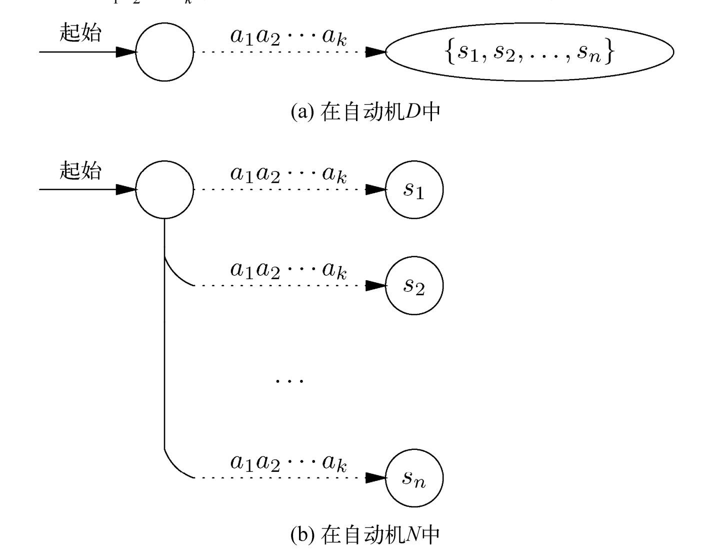


图10-21　非确定自动机N的操作与对应的确定自动机D的操作间存在的关系

我们需要证明图10-21所示的关系，该证明要对输入字符串长度k进行归纳。需要通过对k的归纳来证明的正式命题是，从D的起始状态开始沿着标记为$a_1a_2\cdots a_k$的路径到达的状态$\{s_1, s_2, \cdots, s_n\}$，刚好是从$N$的起始状态开始沿着标记为$a_1a_2\cdots a_k$的路径到达的状态构成的集合。

依据。设$k=0$。长度为0的路径将我们留在出发的位置，也就是，在自动机D和N的起始状态中。回想一下，如果s_0是N的起始状态，D的起始状态就是$\{s_0\}$。因此该归纳命题对$k=0$来说成立。

归纳。假设该命题对k成立，并考虑输入字符串$a_1a_2\cdots a_ka_{k+1}$。那么标记为$a_1a_2\cdots a_ka_{k+1}$的从D的起始状态到状态T的路径就如图10-22所示，也就是说，在它针对输入a_{k+1}进行到T的转换（最后一次转换）之前，会经过某个状态S。

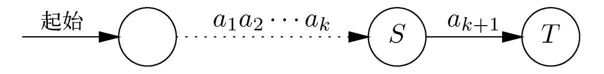


图10-22 S是D在到达状态T之前到达的状态

通过归纳假设，可以假设S正好是自动机N从其起始状态沿着标记为$a_1a_2\cdots a_k$的路径所能到达的状态组成的集合，并必须证明T刚好是从N的起始状态出发沿着标记为$a_1a_2\cdots a_ka_{k+1}$的路径所能到达的状态组成的集合。该归纳步骤的证明包含下列两个部分。

(1)必须证明，T不含过多的状态，也就是说，如果t是在T中的N的状态，那么t是从N的起始状态沿着标记为$a_1a_2\cdots a_ka_{k+1}$的路径可以到达的。

(2)必须证明，T包含足够的状态，也就是说，如果t是从N的起始状态沿着标记为$a_1a_2\cdots a_ka_{k+1}$的路径可以到达的状态，那么t就在T中。

对(1)来说，设t在T中。那么，如图10-23所示，在S中一定存在一个状态s，可以证实t在T中。也就是说，在N中存在从s到t的转换，而且它的标号包含a_{k+1}。根据归纳假设，因为s在S中，所以肯定存在从N的起始状态到s的标记为$a_1a_2\cdots a_k$的路径。因此，存在从N的起始状态到t，标记为$a_1a_2\cdots a_ka_{k+1}$的路径。

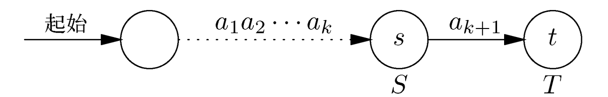


图10-23 S中的状态s解释了为何将状态t放进T中

现在必须证实(2)，也就是如果存在从N的起始状态到t的，标记为$a_1a_2\cdots a_ka_{k+1}$的路径，那么t就在T中。就在这条路径针对输入a_{k+1}进行到t的转换之前，肯定会经过某个状态s。因此，存在从N的起始状态开始到s，标记为$a_1a_2\cdots a_k$的路径。根据归纳假设，s在状态集S中。因为N具有从s到t而且标号含有a_{k+1}的转换，所以应用到状态集S和输入符号a_{k+1}上的子集构造需要t被放置到T中。因此t在T中。

在给定归纳假设的情况下，现在就证明了，T刚好是由N中从N的起始状态开始沿着标记为$a_1a_2\cdots a_ka_{k+1}$的路径可达的状态组成的。这就是归纳步骤，因此可以得出，沿着标记为$a_1a_2\cdots a_k$的路径到达的确定状态机D的状态，永远都是N沿着标号相同的路径可达的状态组成的集合。因为D的接受状态是那些包含N的某个接受状态的状态集，所以可以得到D和N接受相同字符串的结论，也就是说D和N是等价的，所以子集构造是“行得通的”。

10.4.4　习题

(1) 利用子集构造，把10.3节习题(3)中的非确定自动机转换成确定自动机。

(2) 图10-24a到图10-24d中的非确定自动机可以识别什么模式？

(3) 把图10-24a到图10-24d中的非确定自动机转换成确定有限自动机。

自动机的最小化

与自动机相关，特别是在利用自动机设计电路时会遇到的一个问题就是，执行某给定任务需要多少状态。也就是说，我们可能要问，给定某自动机，是否存在状态更少的等价自动机?如果这样的话，那么这样的等价自动机中最小的状态数量是多少?

事实证明，如果将问题限制在确定自动机的范畴，那么与任意给定自动机等价的最小状态确定自动机是唯一的，而且很容易找到它。关键就在于定义确定状态机两个状态s和t什么时候是等价的，也就是说，对任意的输入序列，分别从s和t出发而且由该序列标记的路径，要么都能到达接受状态，要么都不能到达。如果状态s和t是等价的，就没法通过为自动机提供输入来区分这两者，因此就可以把s和t合并为一个状态。事实上，按照如下方式定义不等价的状态要更容易。

依据。如果s是接受状态而t是不接受，那么s和t是不等价的，反之亦然。

归纳。如果存在某输入符号x，使得针对输入x存在从状态s和t分别到两个已知状态的转换不等价，那么s和t就是不等价的。

要让这个测试起效，还需要补充一些细节。特别要提的是，我们可能必须添加一个“停滞状态”，它不接受任何输入，而且针对所有输入都存在到它自身的转换。由于确定自动机可能针对某个给定符号不存在从某给定状态出发的转换，因此在执行这一最小化程序之前，需要针对所有不存在其他转换的输入，添加从任意状态到该停滞状态的转换。可以注意到，不存在类似的用于最小化非确定自动机的理论。

---

(4) * 某些自动机具有一些根本不存在转换的“状态–输入”组合。如果状态s不存在针对符号x的转换，我们就可以添加一种针对符号x的、从s到某个特殊的“停滞状态”的转换。停滞状态是不接受状态，而且针对任意输入符号都有到其自身的转换。证明，添加了“停滞状态”的自动机与原有的自动机是等价的。

(5) 证明，如果为确定自动机添加了停滞状态，就可以得到具有从起始状态出发而且标记为每个可能字符串的路径的等价自动机。

(6) * 证明，如果对确定自动机进行子集构造，那么要么得到相同的自动机，其中每个状态s都重命名为$\{s\}$，要么添加了停滞状态（对应空状态集）。

(7) ** 假设有某确定自动机，并要把每个接受状态变成不接受状态，还要把每个不接受状态变成接受状态。

(a) 如何用旧自动机的语言来描述新自动机接受的语言?

(b) 如果先为原自动机加上停滞状态，重复(a)小题。

(c) 如果原自动机是非确定自动机，重复(a)小题。

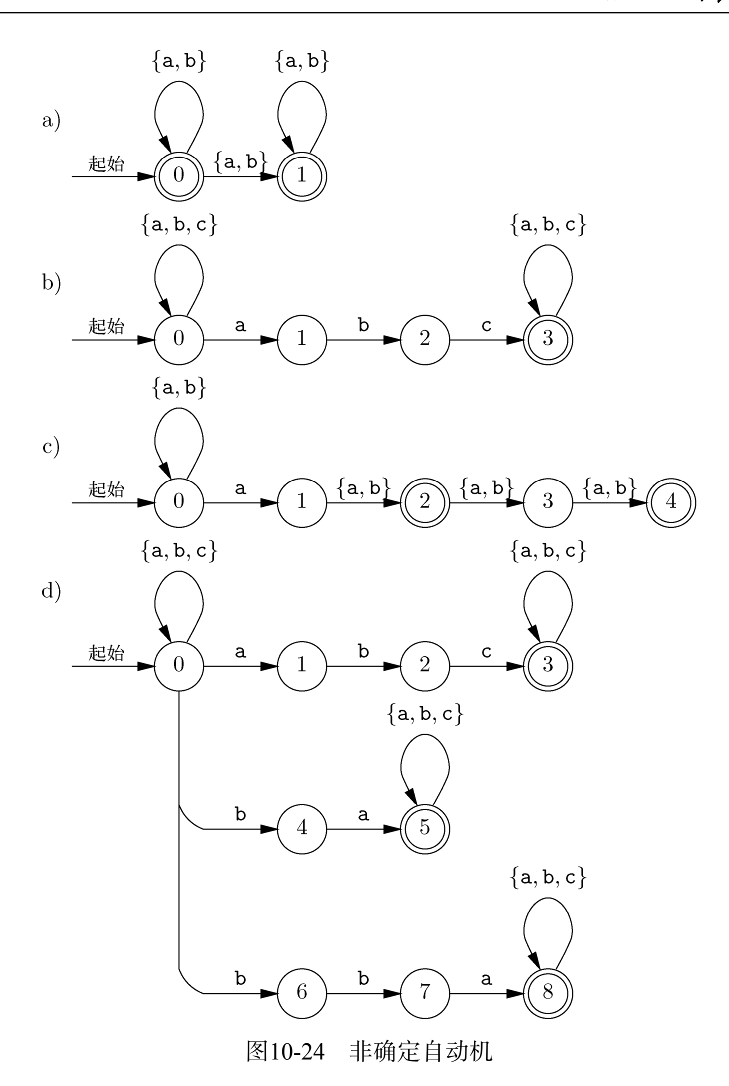


图10-24　非确定自动机

10.5　正则表达式

自动机定义了模式，即表示自动机的图中，作为从起始状态到某个接受状态的路径标号的字符串组成的集合。在本节中，我们遇到了正则表达式这种用来定义模式的代数方法。正则表达式与我们熟悉的算术表达式代数，以及第8章中遇到的关系代数都是相似的。有趣的是，可以用正则表达式代数表示的模式组成的集合，刚好与可以用自动机描述的模式组成的集合相同。

表示方式的约定

我们还将继续使用等宽字体来表示出现在字符串中的字符。与某给定字符对应的正则表达式原子操作数则会用加粗的该字符来表示。例如，**a**是对应字符a的正则表达式。在我们需要使用变量时，会把它写为斜体。变量在这里用来代表复杂的表达式。例如，使用变量*letter*表示“任意字母”这个我们很快就要看到其正则表达式的集合。

10.5.1　正则表达式的操作数

与所有的代数一样，正则表达式也具有某几类原子操作数。在算术代数中，原子操作数是常数（比如整数或实数），或可能的值是常数的变量；而对关系代数而言，原子操作数要么是固定的关系，要么是可能的值为关系的变量。在正则表达式代数中，原子操作数是如下几种中的一种。

(1) 字符；

(2) ϵ 符号；

(3) $\varnothing$ 符号；

(4) 值可能为由正则表达式定义的任意模式的变量。

10.5.2　正则表达式的值

任意代数中的表达式都具有某一类型的值。对算术表达式来说，值可以是整数、实数，或是我们可以处理的任意类型的数字。对关系代数而言，表达式的值就是个关系。

对正则表达式来说，每个表达式的值都是由通常被称为*语言*的字符串集合组成的模式。由正则表达式E表示的语言就被称为$L(E)$，或者是"E的语言"。原子操作数的语言有如下定义。

(1) 如果x是任意字符，那么正则表达式**x**表示语言{x}；也就是说，$L(\mathbf{x})$={x}。请注意，该语言是包含一个字符串的集合，该字符串的长度为1，而且字符串中唯一的位置被字符x占据。

(2) $L(\epsilon)=\{\epsilon\}$。作为正则表达式的特殊字符ϵ表示只含一个空字符串（或者说长度为0的字符串）的字符串集合。

(3) $L(\varnothing)=\{\varnothing\}$。作为正则表达式的特殊字符$\varnothing$表示字符串集合为空。

请注意，我们没有定义原子操作数为变量时的值。只有在将变量替换为具体的表达式时，才可以为这样的操作数取值，而且它的值就是相应的表达式的值。

10.5.3　正则表达式的运算

正则表达式中运算符共有3种。这些运算符都可以用括号分组，就像我们所熟悉的代数中那样。和代数表达式中所做的一样，存在一些让我们可以忽略某些括号对的优先次序和结合律。在探讨完这些运算后，我们将会描述关于括号的规则。

1. 并

第一种，也是我们最熟悉的一种运算符就是*并*运算符，我们要将其表示为|。[①]为正则表达式取并的规则是，如果R和S为两个正则表达式，那么$R|S$表示R和S所表示的语言取并。也就是说，$L(R|S)=L(R)\cup L(S)$。回想一下，$L(R)$和$L(R)$都是字符串的集合，所以为它们取并的概念是说得通的。

✦　示例 10.11

我们知道**a**是表示{a}的正则表达式，而**b**是表示{b}的正则表达式。因此**a|b**就是表示{a，b}的正则表达式。这是一个包含a和b这两个长度为1的字符串的集合，

同样，可以写出诸如(**a|b**)|**c**这样的表达式，来表示集合{a，b，c}。因为取并是一种有结

① 在正则表达式中，加号+也常用作并运算符，不过在这里并不这样表示。

合性的运算符，也就是说，在为3个集合求并集时，以任意次序组合这些操作数都是没关系的，可以忽略括号，并直接将表达式写为**a|b|c**。

2. 串接

正则表达式代数的第二个运算符叫作*串接*（concatenation）。它是用没有任何运算符符号来表示的，就像在写乘法表达式时有时会省略运算符那样，例如，算术表达式ab就表示a和b的积。和取并运算一样，串接运算也是二元的中缀运算符。如果R和S是正则表达式，那么RS就是R和S的串接。[①]

RS表示的语言$L(RS)$是由语言$L(R)$和$L(S)$按照以下方式形成的。对$L(R)$中的各字符串r以及$L(S)$中的各字符串s来说，字符串r和s的串接rs是在$L(RS)$中的。回想一下两个表（比如字符串）的串接，是通过按次序取第一个表中的元素，并在它们之后按次序接上第二个表的元素而形成的。

类型之间的一些微妙区别

读者不应该弄混看似相似，实则差别巨大的多种对象。例如，空字符串ϵ就和空集$\varnothing$不同，而这两者又都与只包含空字符串的集合$\{\epsilon\}$不同。空字符串的类型是“字符串”或“字符表”，而空集和只含空字符串的集合都是“字符串集合”类型的。

我们还应该记得类型为“字符”的字符a，类型为“字符串”的长度为1的字符串a，以及类型为“字符串集合”的正则表达式**a**的值{a}之间的区别。还要注意到在其他的上下文中，{a}可能表示包含字符a的集合，而且我们没有表示方式上的约定用来区分{a}的这两种含义。不过，在本章的内容中，{a}通常都具有前一种解释，也就是“字符串集合”而非“字符集合”。

✦ 示例 10.12

设R是正则表达式**a**，因此$L(R)$就是集合{a}。再设S是正则表达式**b**，所以$L(S)$ = {b}。那么RS就是表达式**ab**。要形成$L(RS)$，需要取$L(R)$中的每个字符串，将其与$L(S)$中的每个字符串串接。在这种简单的情况中，$L(R)$和$L(S)$都是单元素集，所以对其二者来说都各自只有一种选择。我们从$L(R)$中选择a，并从$L(S)$中选择b，然后将这两个长度为1的表串接起来，就得到了字符串ab。因此$L(RS)$就是{ab}。

可以对示例10.12进行概括，得出任意用粗体表示的字符串，都是表示由一个字符串（相应字符组成的表）构成的语言的正则表达式。比如，**then**就是语言为{then}的正则表达式。我们将看到，串接是一种具有结合性的运算符，所以不管正则表达式中的字符如何分组都是没关系的，而且不需要使用括号。

✦ 示例 10.13

现在来看看两个语言不是单元素集的正则表达式的串接。设R是正则表达式**a|(ab)**。[②]语言$L(R)$就是$L(\mathbf{a})$和$L(\mathbf{ab})$的并集，即{a，ab}。设S是正则表达式**c|(cb)**。同样，$L(S)$ = {c，cb}。正

---

① 严格地讲，应该把RS写为$(R)(S)$，以强调R和S是分开的表达式，因为优先级规则，它们各自的组成部分一定不能混合在一起。这种情况与我们将表达式$w+x$与$y+z$相乘时类似，一定要将积写为$(w+x)(y+z)$。请注意，因为乘法要先于加法计算，所以如果把括号去掉，得到的表达式$w+xy+z$就不能解释为$w+x$与$y+z$的积了。正如我们将要看到的，串接与取并具有的优先关系使得它们分别类似与乘法和加法。

② 正如接下来将要看到的，串接要优先于取并，所以这里的括号是多余的。

则表达式RS就是(**a**|(**ab**))(**c**|(**cb**))。请注意，出于运算优先级的原因，R和S上的括号是必要的。

	c	**bc**
a	ac	abc
ab	abc	abbc

图10-25　形成{a，ab}和{c，cb}的串接

要找出$L(RS)$中的字符串，就要将$L(R)$中的两个字符串与$L(S)$中的两个字符串一一配对。这一配对方式如图10-25所示。从$L(R)$中的a和$L(S)$中的c，可以得到字符串ac。而字符串abc可以用两种不同方式得到，要么是(a)(bc)，要么是(ab)(c)。最后，$L(R)$中的ab与$L(S)$中的bc串接就得到字符串abbc。因此$L(RS)$就是{ac，abc，abbc}。

请注意，语言$L(RS)$中的字符串数量不可能大于$L(R)$中字符串数量和$L(S)$中字符串数量的积。事实上，$L(RS)$中字符串的数量刚好就是这个积，除非存在一些"巧合"，也就是同一字符串可以通过两种或多种不同方式形成的情况。示例10.13就是这样一个例子，其中字符串abc就可以用两种方式生成，因此$L(RS)$中就只有3个字符串，要比R和S的语言中字符串数量之积少1。同样，语言$L(R \mid S)$中字符串的数量也不大于语言$L(R)$和$L(S)$中字符串数量的和，而且在$L(R)$和$L(S)$中有相同字符串时只可能比该和小。在讨论这些运算符的代数法则时我们还将看到，正则表达式的取并和串接，与算术运算符＋和×之间，存在一种相近但不精确的类比。

3. 闭包

第三个运算符是*克林闭包*（Kleene closure）或直接称为*闭包*。①它是个一元的后缀运算符，也就是说，它接受一个操作数并且出现在该操作数之后。闭包是用星号表示的，所以R*是正则表达式R的闭包。因为闭包运算符有着最高的优先级，所以通常需要在R两侧放上括号，将其写为(R)*。

闭包运算符的作用是表示"R中的字符串没有出现或多次出现"。也就是说，$L(R*)$由下列内容组成。

(1) 空字符串ϵ，可以将其视作R中的字符串没有出现。

(2) 在$L(R)$中的所有字符串，表示$L(R)$中的字符串出现一次。

(3) 在$L(RR)$中的所有字符串，也就是$L(R)$与自身的串接，表示$L(R)$中的字符串出现两次。

(4) 在$L(RRR)$、$L(RRRR)$等中的所有字符串，表示$L(R)$中的字符串出现3次、4次和更多次。

可以有如下非正式的表示

$$R^{*}=\epsilon \mid R \mid RR \mid RRR \mid \cdots$$

不过，一定要理解，等号右侧的表达式并不是正则表达式，因为它包含无数个取并运算符。而所有正则表达式都是用有限数量的这3种运算符构建的。

✦ 示例 10.14

设R = **a**。那么$L(R*)$是什么？当然，ϵ肯定在该语言中，因为它一定在任意闭包中。而$L(R)$中唯一的字符串a也在该语言中，还有$L(RR)$中的aa，$L(RRR)$中的aaa，等等。也就是说，L(**a***)是由含0个或多个a的字符串组成的集合，也就是{ϵ，a，aa，aaa，…}。

① StevenC.Kleene最早撰写了描述正则表达式代数的论文。

✦ 示例 10.15

现在设R是正则表达式**a|b**，那么$L(R)=\{$a，b$\}$，并考虑$L(R^*)$是什么。该语言还是含有ϵ，表示$L(R)$中字符串没有出现。R中的字符串出现一次就为$L(R^*)$带来了{a，b}。出现两次就给我们4个字符串{aa，ab，ba，bb}，3次出现就给了我们由a和（或）b组成长度为3的8个字符串，以此类推。因此$L(R^*)$是所有由a和（或）b组成的任意有限长度的字符串。

10.5.4　正则表达式运算符的优先级

正如我们在前面的内容中非正式提到的，正则表达式的3个运算符并、串接和闭包之间存在约定的优先级次序。这一次序如下。

(1) 闭包（最高）；

(2) 然后是串接；

(3) 然后是并（最低）。

因此，在解释任何正则表达式时，首先要为闭包运算符分组，也就是找出具有表达式形式（即如果存在括号的话，则它们是配对的）的紧邻某给定*左侧的最短表达式。可以给该表达式和相应的*加上一对括号。

接下来，要从左边起考虑串接运算符。对每个串接运算符，要找到紧邻其左侧的最小表达式并找到紧邻其右侧的最小表达式，再给这一对表达式加上括号。最后，要从左侧起考虑取并运算符。找到紧邻每个取并运算符左右的表达式，并这这一对中间有着取并符号的表达式周围加上括号。

✦ 示例 10.16

考虑表达式**a|bc*d**。首先分析*。该表达式中只有一个*，而且在其左侧的最小表达式为**c**。因此可以把该*与他的操作数分到一组，就成了**a|b(c*)d**。

接下来，要考虑上述表达式中的串接。共有两次串接，一次是**b**和左括号之间的，另一次是在右括号和**d**之间的。首先分析第一次串接，我们看到**b**就是紧邻左侧的，而到右侧就必须到将右括号包括在内为止，因为表达式的括号必须是平衡的。因此，第一次串接的操作数分别是**b**和(**c***)。给它们周围加上括号就得到表达式

$$\mathrm{a}\,|\,\big(\mathrm{b}(\mathrm{c}*)\big)\mathrm{d}$$

对第二次串接来说，紧邻其左的最短表达式现在是$\big(\mathrm{b}(\mathrm{c}*)\big)$，而紧邻其右的最短表达式是**d**。在给这次串接的操作数分组加上括号后，表达式就成了

$$\mathrm{a}\,|\,\Big(\big(\mathrm{b}(\mathrm{c}*)\big)\mathrm{d}\Big)$$

最后，必须考虑取并运算。该表达式中总共有一次取并运算，它的左操作数为**a**，而其右操作数就是上述表达式其余的部分。严格来说，必须为整个表达式再加上一层括号，得到

$$\Big(\mathrm{a}\,|\,\big(\mathrm{b}(\mathrm{c}*)\mathrm{d}\big)\Big)$$

不过最外层的括号是多余的。

10.5.5　正则表达式的其他一些示例

在本节最后，我们要给出一些更复杂的正则表达式。

✦ 示例 10.17

可以将示例10.15中的思路扩展到“由符号a_1、a_2、…、a_n组成的任意长度的字符串”以及如下正则表达式：

(a1|a2|…|an)*

例如，可以按照以下方式描述C语言标识符。首先，定义正则表达式：

letter = **A**|**B**|…|**Z**|**a**|**b**|…|**z**|_

这就是说，C语言中的“字母”包括大小写英文字母和下划线。同样，我们还定义了正则表达式：

digit = 0|1|… | 9

那么正则表达式

letter(*letter*|*digit*)*

就表示由字母、数字和下划线组成的所有不以数字开头的字符串。

✦ 示例 10.18

现在来考虑一个更难写出的正则表达式：在示例10.2中讨论过的反弹过滤器问题。回想一下，我们描述了这样一种自动机，粗略地讲，就是只要输入的结尾是一列1，就会输出1。也就是说，只要在一行中看到两个1，就认为我们已经在一列1中，不过在确定看到的是一列1后，一个0的出现并不会让我们推翻这一结论。因此，每当输入的结尾有由两个1组成的序列时，只要它后面跟上的内容中每个0之后要么立即跟上一个1，要么是当前为止看到的最后一个输入字符，示例10.2中自动机的输出就是1。可以用如下正则表达式表示这种情况。

(**0**|**1**)***11**(**1**|**01**)*(ϵ|0)

要理解该正则表达式，首先要注意到(**0**|**1**)*表示由0和1组成的任意字符串。这些字符串后面一定要跟上两个1，就如表达式**11**所表示的。因此(**0**|**1**)***11**就是所有由0和1组成且结尾（至少）有两个1的字符串。

接下来(**0**|**01**)*表示所有由0和1组成，而且其中所有0后面都跟着1的字符串。也就是，该表达式语言中的字符串是以任意次序串接任意数量的字符串1和01构成的。尽管**1**让我们在任意时刻都可以向正在形成的字符串添加一个1，不过**01**强迫我们在任何0之后都加上一个1。因此表达式(**0**|**1**)***11**(**1**|**01**)*表示所有由0和1组成的，以两个1后面加上其中的0都要立即加上一个1的任意序列结尾的字符串。而最后的因子(ϵ|**0**)表示“可选的0”，也就是说，刚刚描述的字符串后面可能还有一个0，也可能没有，要看我们的选择。

10.5.6　习题

(1) 在示例10.13中，我们描述了正则表达式(**a**|**ab**)(**c**|**cb**)，并看到它的语言是由ac、abc和abbc这3个字符串组成的，也就是说，一个a和一个c，中间被0到2个b分隔开。再写两个定义该语言的正则表达式。

(2) 写出定义下列语言的正则表达式。

  (a) 对应6个C语言比较运算符=、<=、<、>=、>和!=的字符串。
  (b) 所有由0和1组成且结尾为0的字符串。
  (c) 所有由0和1组成且至少有一个1的字符串。
  (d) 所有由0和1组成且至多有一个1的字符串。
  (e) 所有由0和1组成且至右起第三位是1的字符串。
  (f) 所有由小写字母按已排序次序组成的字符串。

(3) * 写出定义下列语言的正则表达式。

(a) 所有由a和b组成，满足其中由a组成的子序列长度都是偶数的字符串。也就是说，诸如bbbaabaaaa、aaaabb和ϵ这样的字符串，而abbabaa和aaa就不是。

(b) 由C语言中float类型的数字表示的字符串。

(c) 由0和1组成且具有偶校验（即含偶数个1）的字符串。提示：将偶校验字符串视作具有偶校验的基本字符串（要么是一个0，要么是只由0分隔的一对1）的串接。

(4) ** 写出定义下列语言的正则表达式。

(a) 不是关键字的C语言标识符组成的集合。如果忘记了某些关键字也是没关系的，本题的重点在于表示那些不在某个相当大的字符串集合中的字符串。

(b) 所有由a、b和c组成，而且满足任何两个连续位置上的字母都不相同的字符串。

(c) 由两个不同的小写字母形成的所有字符串构成的集合。提示：大家可以用“蛮力”来解决问题，不过用两个不同的字母构成的字母对有650个。更好的思路是进行一些分组。例如，相对较短的表达式(**a|b|**…**|m**)(**n|o|**…**|z**)就能覆盖这650个字母对中的169个。

(d) 所有由二进制数字0和1组成的，表示为3的倍数的整数的字符串。

(5) 根据取并、串接和闭包运算符的优先级，为以下正则表达式加上括号，以表示操作数的恰当分组。

(a) **a|bc|de**

(b) **a|b***|(**a|b**)***a**

(6) 从下列正则表达式中删除多余的括号，也就是说，删除那些分组可以由运算符的优先级以及取并和串接的结合性（因此为邻接的取并或邻接的串接分组是无关紧要的）暗示出的括号。

(a) (**ab**)(**cd**)

(b) **(a|(b(c)*))**

(c) **(((a)|b(c|d))**

(7) * 描述由以下正则表达式定义的语言。

(a) $\emptyset|\epsilon$

(b) ϵ **a**

(c) (**a|b**)*

(d) (**a*b***)*

(e) (**a*ba*b**)***a***

(f) ϵ*

(g) R**，其中R是任意正则表达式。

10.6 UNIX对正则表达式的扩展

UNIX操作系统中有不少命令利用了类似正则表达式的表示法来描述模式。即便对UNIX操作系统与其中的大部分命令并不熟悉，了解这些表示法也还是很实用的。我们发现正则表达式至少用在如下3类命令中。

(1) 编辑器。UNIX编辑器ed和vi，以及大多数现代文本编辑器，让用户可以在找到某给定模式实例的位置扫描文本。这一模式是由正则表达式指定的，虽然没有一般的取并运算符，只有下面将要讨论的“字符类”。

(2) 模式匹配程序grep及类似程序。UNIX命令grep会对文件进行扫描，检查文件的每一行。如果该行包含某个能与由正则表达式指定的模式匹配的子串，就将该行打印出来（grep代表globally search for regular expression and print，即全局查找正则表达式并打印）。grep命令本身只接受正则表达式的子集，而扩展的命令egrep则可以接受完整的正则表达式表示，而且包

含了一些其他的扩展。命令awk允许我们进行全面的正则表达式搜索，并且把文本行当作关系的元组来处理，从而使我们可以对文件执行选择和投影这样的关系代数运算。

(3) 词法分析。UNIX命令lex对编写编译器代码及类似任务而言是很使用的。编译器第一件必须完成的事就是将程序分割为一个个标记（token），它们是逻辑上结合在一起的子字符串。标记的例子包括标识符、常量、then这样的关键字，以及+或<=这样的运算符。每种标记类型都可以由一个正则表达式来指定，比方说，示例10.17就展示了如何指定“标识符”标记类。lex命令让用户可以用正则表达式指定标记类。这样就形成了可以用作词法分析器的程序，也就是可以把输入分解为标记的程序。

10.6.1　字符类

我们经常需要写出表示字符集合，或者严格地讲，是表示长度为1的字符串（每个字符串都是由集合中不同的字符构成）组成的集合的正则表达式。因此，在示例10.17中，我们定义了表达式 *letter*表示任何由一个大写字母或小写字母组成的字符串，并定义了表达式 *digit*表示任何由一个数字构成的字符串。这些表达式都是相当长的，而UNIX提供了一些重要的简写。

首先，可以用方括号把任意字符表括起来，用来代表对这些字母取并的正则表达式。这样的表达式就叫作字符类。例如，表达式[**aghinostw**]表示出现在单词washington中的字母组成的集合，而[**aghinostw**]*则表示只由这些字母形成的字符串所构成的集合。

其次，我们并非总是需要明确地列出所有的字符。回想一下，字母几乎一直都是用ASCII编码的。这种编码会为各种字符指定相应的位串（很自然地就可以解释为整数），而且它是以一种合理的方式来完成这一工作的。例如，ASCII编码为大写字母分配了连续的整数。同样，它也为小写字母以及数字分配了连续的整数。

如果在两个字符间加上破折号，就不仅表示这些字符，而且表示了编码在这两个字符编码之间的所有字符。

✦　示例 10.19

我们可以通过[**A-Za-z**]定义大写字母与小写字母。前3个字符**A-Z**表示编码处于A和Z之间的所有字符，也就是所有的大写字母。而接下来的3个字符**a-z**则表示所有的小写字母。

顺便提一句，因为破折号有这样一种特殊含义，所以如果想要定义包含-的字符类，就一定要谨慎行事。必须把破折号放在这列字符的第一个位置或最后一个位置。例如，可以通过**[-+*/]**来指定4四种算术运算符组成的集合，但如果写成**[+-*/]**的形式就是错误的，因为+-*这样的范围会表示编码在+和*的编码之间的所有字符。

10.6.2　行的开头和结尾

因为UNIX命令经常要处理单行文本，所以UNIX正则表达式表示法中包含了用于表示行的开头和结尾的特殊符号。符号^表示行的开头，而$表示行的结尾。

✦　示例 10.20

10.3节图10-12中的自动机是从一行的开头处启动的，它接受的文本行刚好是那些只由单词washington中的字母组成的文本行。可以将这种模式表示为UNIX正则表达式：**^[aghinostw]*$**。口头上讲，该模式就是“行的开头，后面跟上由单词washington中的字母组成的任意序列，再加上行的结尾”。

举个这种正则表达式使用方式的例子，UNIX命令行：

```
grep'^[aghinostw]*$ /usr/dict/words
```

将打印出词典中所有只由来自washington的字符组成的单词。在这种情况下，UNIX要求该正则表达式被写为引用的字符串。这一命令的效果就是指定的文件/usr/dict/words中每一行都会被检查。如果它含有任何处在由该正则表达式表示的字符串集合中的子字符串，那么这一行就要被打印出来，否则这一行就不会被打印。请注意，这一行的开头符号与结尾符号已然存在了。假设它们不存在。因为空字符串是在由正则表达式 **[aghinostw]***表示的语言中的，所以我们会发现每一行都有一个子字符串（即ϵ）位于该正则表达式的语言中，因此每一行都会被打印出来。

为字符赋予字面意义

顺便说一下，因为字符^和$在正则表达式中被赋予了特殊意义，所以看起来没办法在UNIX正则表达式中指定这些字符本身了。不过，UNIX用到了反斜杠\，作为转义字符。如果我们在字符^或$之前加上反斜杠,那么这两个字符形成的组合就会被解释为第二个字符的字面意义，而不是其特殊含义。例如\$表示UNIX正则表达式中的$字符。同样，两道反斜杠就被解释为一个反斜杠，而不含有其转义字符的特殊意义。而UNIX正则表达式中的字符串\\$表示的是反斜杠字符后面跟上行的结尾。

还有不少其他的字符也被UNIX在某些情形下赋予了特殊意义，而这些字符也总是能表示它们的字面意义，也就是，通过在它们之前使用反斜杠来“除掉”它们的特殊意义。例如，只有这样处理方括号，方括号在UNIX正则表达式中才不会被解释为字符类分隔符。

10.6.3 通配符

符号.在UNIX正则表达式中代表“除换行符之外的任意字符”。

✦ **示例 10.21**

正则表达式

```
.*a*e.*o.*u.
```

表示按次序包含5个元音字母的所有字符串。我们可以利用grep与该正则表达式来扫描词典，查找单词中5个元音字母按递增次序出现的所有单词。不过，如果忽略掉开头和结尾位置的.*，处理将更具效率，因为grep是按子字符串搜索指定的模式，而不是整行搜索，除非我们显式地包含了表示行开头和行结尾的符号。因此命令

```
grep'a.*e.*i.*u' /use/dict/words
```

将会找到含有子序列aeiou的所有单词并将其打印出来。

这些点号会匹配除字母之外的字符，这一实情并不重要，因为在/usr/dict/words文件中除了字符和换行符之外没有其他字符。不过，如果点号可以匹配换行符，那么这一正则表达式就会允许grep一次使用多行来找出依次出现的5个元音字母。不过像本例这样的例子都是点号被定义为不匹配换行符的例子。

10.6.4 额外的运算符

UNIX命令awk和egrep中的正则表达式还含有一些额外的运算符，具体如下。

(1) 与grep不同的是，awk和egrep命令还允许取并运算符|出现在它们的正则表达式中。

(2) 一元后缀运算符?和$^+$没有允许我们定义额外的语言，但它们通常会让表示语言的工作变得更简单。如果R是正则表达式，则R?代表$\epsilon \mid R$，也就是可选的R。所以$L(R?)$是$L(R) \cup \{\epsilon\}$。R^+代表RR^*，或者等价地讲就是"R中的单词出现一次或多次"。因此，$L(R^+) = L(R) \cup L(RR) \cup L(RRR) \cdots$。特别要说的是，如果$\epsilon$在$L(R)$中，那么$L(R^+)$和$L(R^*)$表示相同的语言。而如果$\epsilon$不在$L(R)$中，那么$L(R^+)$就表示$L(R^*) - \{\epsilon\}$。运算符?和$^+$与*有着相同的结合性与优先级。

✦ 示例 10.22

假设我们想通过正则表达式来指定由非空数字串与一个小数点组成的实数。将该表达式写为[0-9]*\.[0-9]*是不正确的，因为这样一来，只由一个点号组成的字符串也会被视作实数。该表达式利用egrep的一种写法是

[0-9]$^+$\.[0-9]*\.[0-9]$^+$

在这里，取并的第一项涵盖了那些小数点左侧至少有一个数字的实数，而第二项则涵盖了以小数点开头，因此在小数点后必须至少有一位数字的那些实数。请注意，放在点号之前的反斜杠是为了表明这里的点号不具有约定的"通配符"含义。

✦ 示例 10.23

利用如下egrep命令可以扫描输入中那些字母严格按照字母表增序排列的行。

```
egrep '^a?b?c?d?e?f?g?h?i?j?k?l?m?n?o?p?q?r?s?t?u?v?w?x?y?z?$'
```

也就是说，我们会扫描每一行，看看在行的开头和结尾之间是否有可选的a、可选的b，等等。例如，含有单词adept的一行就能匹该表达式，因为a、d、e、p和t之后的?可以解释为"出现一次"，而其他的?可以解释为"没有出现"，也就是ϵ。

10.6.5 习题

(1) 为以下字符类写出表达式。

 (a) 所有属于C语言运算符和标点符的字符，例如+和圆括号。

 (b) 所有小写元音字母。

 (c) 所有小写辅音字母。

(2) * 如果可以使用UNIX，编写egrep程序检查/usr/dict/words文件，并找到下列单词：

 (a) 所有以dous结尾的单词；

 (b) 所有只含一个元音字母的单词；

 (c) 所有原音字母与辅音字母交替出现的单词；

 (d) 所有含四个或更多个连续辅音字母的单词。

10.7 正则表达式的代数法则

两个正则表达式是可以表示同一语言的，就像两个算术表达式可以表示其操作数的相同函数那样。举例来说，$x+y$和$y+x$这两个表达式就表示x和y的相同函数。同样，不管用什么正则表

达式来替换R和S，正则表达式$R|S$和$S|R$都表示同一语言，证据就是取并运算也是具有交换性的。

简化正则表达式往往是很实用的。我们很快就会看到，在根据自动机构造正则表达式时，经常会构造出过于复杂的正则表达式。代数等价可以让我们“简化”表达式，也就是说，把一个正则表达式替换为另一个操作数和（或）运算符更少却又表示相同语言的正则表达式。这一过程类似于在处理算术表达式时对繁冗的表达式进行的那些简化。例如，可以将两个很大的多项式相乘，然后通过分组相似项来简化结果。再比如，我们在8.9节中简化了关系代数表达式从而获得更快的求值速度。

如果$L(R)=L(S)$，就说两个正则表达式R和S是等价的，记作$R \equiv S$。如果这样的话，可以说$R \equiv S$是一种等价。在接下来的内容中，我们将假设R、S和T是任意的正则表达式，并以这些操作数来陈述要讨论的等价关系。

等价的证明

在本节中，我们要证明若干涉及正则表达式的等价关系。回想一下，两个正则表达式的等价是说，不管为其变量替换怎样的语言，这两个表达式的语言都是相等的。因此我们可以通过证明两种语言，也就是两个字符串集合的相等性，来证明正则表达式的等价。一般而言，要通过证明两个方向上的包含关系来证明集合S_1和集合S_2是等价的。也就是说，证明$S_1 \subseteq S_2$，还要证明$S_2 \subseteq S_1$。两个方向对证明集合相等都是必要的。

10.7.1　取并和串接与加法和乘法的类比

在本节中，我们要列举与正则表达式的取并、串接和闭包运算符有关的最重要的那些等价关系。首先要从取并和串接运算与加法和乘法运算的类比开始。正如我们将要看到，这种类比并不是很精确，主要因为串接是不具备交换性的，而乘法当然是具备交换性的。不过，这两对运算之间还是存在诸多相似性。

首先，取并和串接都具有单位元。取并运算的单位元是$\varnothing$，而串接运算的单位元是ϵ。

(1) 取并的单位元。$(\varnothing \mid R) \equiv (R \mid \varnothing) \equiv R$。

(2) 串接的单位元。$\epsilon R \equiv R\epsilon \equiv R$。

从空集和取并的定义中应该不难看出(1)成立的原因。要知道(2)成立的原因，如果字符串x在$L(\epsilon R)$中，那么x就是$L(\epsilon)$中某个字符串与$L(R)$中某个字符串r的串接。不过$L(\epsilon)$中唯一的字符串就是ϵ本身，因此可知$x=\epsilon r$。不过，空字符串与任意字符串r串接的结果都是r本身，所以$x=r$。也就是说x在$L(R)$中。同样可以看到，如果x在$L(R\epsilon)$中，那么x也在$L(R)$中。

要证明等价对(2)，不仅要证明$L(\epsilon R)$和$L(R\epsilon)$中的每个字符串都在$L(R)$中，而且要证明反向的命题，也就是$L(R)$中的每个字符串都在$L(\epsilon R)$和$L(R\epsilon)$中。如果r在$L(R)$中，那么ϵr就在$L(\epsilon R)$中。不过$\epsilon r = r$，所以r在$L(\epsilon R)$中。同样的推理告诉我们r也在$L(R\epsilon)$中。因此就证明了$L(R)$和$L(\epsilon R)$是相同的语言，而且$L(R)$和$L(R\epsilon)$是相同的语言，这就是(2)中的等价关系。

因此$\varnothing$与算术中的0是类似的，而ϵ则与1是类似的。还存在另一种类比，$\varnothing$是串接的零元（annihilator），也就是

(3) 串接的零元。$\varnothing R \equiv R\varnothing \equiv \varnothing$。换句话说，当将空集与任何内容串接时，得到的都是空集。类似地，0是乘法运算的零元，因为$0 \times x = x \times 0 = 0$。

可以通过以下方式验证(3)的真实性。为了让某个字符串x出现在 $L(\varnothing R)$ 中，就要用 $L(\varnothing)$ 中的字符串串接$L(R)$中的字符串。因为 $L(\varnothing)$ 中是不含字符串的，所以我们没办法形成x。类似的论证也可以证明 $L(R\varnothing)$ 也必定为空。

接下来的等价关系是我们在第7章中讨论过的并集运算的交换律和结合律。

(4) 取并的交换律。$(R\,|\,S)\equiv(S\,|\,R)$。

(5) 取并的结合律。$((R\,|\,S)|\,T)\equiv(R|(S\,|\,T))$。

正如我们提过的，串接也是有结合性的。也就是

(6) 串接的结合律。$((RS)\,T)\equiv(R(ST))$。

要知道(6)为何成立，先假设字符串x在 $L((RS)T)$ 中，那么x就是由$L(RS)$中的某字符串y和$L(T)$中某字符串t串接而成。还有，y一定是由$L(R)$中的某字符串r和$L(S)$中的某字符串s串接而成，因此有 $x=yt=rst$。现在考虑 $L(R(ST))$。字符串st一定在$L(ST)$中，因此rst，也就是x，是在 $L(R(ST))$ 中的。因此 $L((RS)\,T)$ 中的每个字符串x都一定也在 $L(R(ST))$ 中。相似的论证告诉我们 $L(R(ST))$ 中的每个字符串也一定在 $L((RS)\,T)$ 中。因此这两种语言是相同的，所以等价关系(6)成立。

(7) 串接对取并的左分配律。$(R(S|\,T))\equiv(RS\,|\,RT)$。

(8) 串接对取并的右分配律。$((S\,|\,T)R)\equiv(SR\,|\,TR)$。

我们看看(7)为什么成立，(8)成立的原因也是相似的，就留作习题吧。如果x在 $L(R(S\,|\,T))$ 中，那么 $x=ry$，其中r在$L(R)$中，而且y要么在$L(S)$中，要么在$L(T)$中，或者同在这两者中。如果y在$L(S)$中，那么x在$L(RS)$中，而如果y在$L(R)$中，那么x在$L(RT)$中。不管是哪种情况，x都是在$L(RS|RT)$中。因此 $L(R(S\,|\,T))$ 中的每个字符串都在$L(RS|RT)$中。

我们还可以证明反向的命题，也就是说 $L(RS|RT)$中的每个字符串都在 $L(R(S\,|\,T))$ 中。如果x在前者中，那么x要么在$L(RS)$中，要么在$L(RT)$中。假设x在$L(RS)$中。那么$x = rs$，其中r在$L(R)$中，s在$L(S)$中。因此s在$L(S|T)$中，所以x在 $L(R(S\,|\,T))$ 中。同样，如果x在$L(RT)$中，就可以证明x一定在 $L(R(S\,|\,T))$ 中。现在我们就证明了两个方向上的包含，这样就证明了等价关系(7)。

10.7.2　取并和串接与加法和乘法的区别

取并运算与加法不尽相同的原因之一就是幂等律。也就是说，取并运算是幂等的，但加法不是。

(9) 取并的幂等律。$(R\,|\,R)\equiv R$。

串接也与乘法有重大差异，因为串接不具交换性，而实数或整数的乘法是满足交换律的。要知道RS一般来说与SR不等价的原因，可以举个简单例子，设$R=$ a而且$S=$ **b**，则有$L(RS)=$ {ab}，且$L(RS)=$ {ba}，而这两个集合是不同的。

10.7.3　涉及闭包的等价

还有其他一些与闭包运算符有关的实用等价关系。

(10) $\varnothing^*\equiv\epsilon$。大家可以验证该式的两边都表示语言$\{\epsilon\}$。

(11) $RR^*\equiv R^*R$。请注意，该式两边都与10.6节扩展表示法中的R^+是等价的。

(12) $(RR^*|\,\epsilon)\equiv R^*$。也就是说，$R^+$和空字符串取并是与$R^*$等价的。

10.7.4 习题

(1) 证明等价关系(8)，即串接对取并的右分配律是成立的。

(2) 通过变量替换，可以从已经陈述的等价关系得出等价关系 $\varnothing\varnothing \equiv \varnothing$ 和 $\epsilon\epsilon \equiv \epsilon$，其中要利用到哪些等价关系？

(3) 证明等价关系(10)到(12)。

(4) 证明：

(a) $(R \mid R^*) \equiv R^*$；

(b) $(\epsilon \mid R^*) \equiv R^*$。

(5) * 是否存在特定的正则表达式R和S满足“交换律”，也就是说，对这些特殊的表达式来说，满足$RS = SR$？如果不存在，请给出证明，如果存在，请给出示例。

(6) * 正则表达式中不需要操作数 $\varnothing$，除非没有 $\varnothing$ 就找不到语言为空集的正则表达式。如果正则表达式中未出现 $\varnothing$，就说它是无 $\varnothing$ 的。通过对无 $\varnothing$ 正则表达式R中出现的运算符个数进行归纳，证明$L(R)$不是空集。提示：10.8节中将会展示对正则表达式中出现的运算符的数量进行归纳证明的示例。

(7) ** 通过对正则表达式R中出现的运算符的数量进行归纳证明：R要么与正则表达式 $\varnothing$ 等价，要么与某个无 $\varnothing$ 正则表达式等价。

10.8 从正则表达式到自动机

记得我们在10.2节中对自动机的初步讨论，其中看到了在确定自动机与利用了“状态”概念（用以区分程序中不同部分扮演的角色）的程序之间存在的紧密关系。然后我们说过，设计确定自动机往往是设计这类程序的一种好方法。不过我们还看到，确定自动机可能难于设计。在10.3节中看到，有时候设计非确定自动机要更简单，而且子集构造让我们可以把任意非确定自动机转换成等价的确定自动机。现在我们又知道了正则表达式，接下来会看到，写出正则表达式甚至比设计非确定自动机更简单。

而且很好的是，存在某种方式可以把任意正则表达式转换成非确定自动机，接着就可以使用子集构造将得到的非确定自动机转换成确定自动机。事实上，我们在10.9节中还会看到，也能把任意自动机转换成相应的正则表达式，其中正则表达式的语言刚好是自动机接受的字符串集合。因此自动机和正则表达式有着一模一样的语言描述能力。

并非所有语言都能用自动机描述

尽管我们看到很多语言可以用自动机或正则表达式描述，但还是存在不能这样描述的语言。直觉就是“自动机不会数数”。也就是说，如果为具有n个状态的自动机提供一列n个相同符号，它肯定会进入同一状态两次，这样它就不能确切记住已经看到了多少符号。因此，比方说自动机不可能识别所有只由平衡圆括号组成的字符串。因为正则表达式和自动机定义了相同的语言，所以同样不会有语言刚好全是平衡圆括号字符串的正则表达式。我们将在10.9节中讨论哪些语言是不可以用自动机定义的。

在本节中，我们要完成以下任务，说明如何把正则表达式转换成自动机。

(1) 引入具有 ϵ 转换的自动机，也就是，具有标号为 ϵ 的弧的自动机。这些弧用在路径中，

但不会对路径的标号产生任何影响。这种形式的自动机是正则表达式与本章先前讨论过的自动机之间的中间体。

(2) 展示如何把任意正则表达式转换成定义同一语言的具有 ϵ 转换的自动机。

(3) 展示如何把任意具有 ϵ 转换的自动机转换成接受同一语言的不含 ϵ 转换的自动机。

10.8.1　具有 ϵ 转换的自动机

首先要将自动机的概念扩展到允许为弧标记 ϵ 。这样的自动机仍然是当且仅当从起始状态到接受状态存在标记为 s 的路径时才接受字符串 s。不过请注意，空字符串 ϵ 在字符串中是“不可见的”，因此在为路径构建标号时，我们其实删除了所有的 ϵ ，并且只使用“真实”的字符。

✦　示例 10.24

考虑如图10-26所示的具有 ϵ 转换的自动机。在这里，状态0是起始状态，而状态3是唯一的接受状态。从状态0到状态3的一条路径为

0，4，5，6，7，8，7，8，9，3

这些弧的标号就构成了序列

ϵ b ϵ ϵ c ϵ c ϵ ϵ

只要记得 ϵ 与任意其他字符串串接得到的都是那个其他字符串，就可以“丢掉”这些 ϵ 得到字符串bcc，这就是正在考虑的路径的标号。

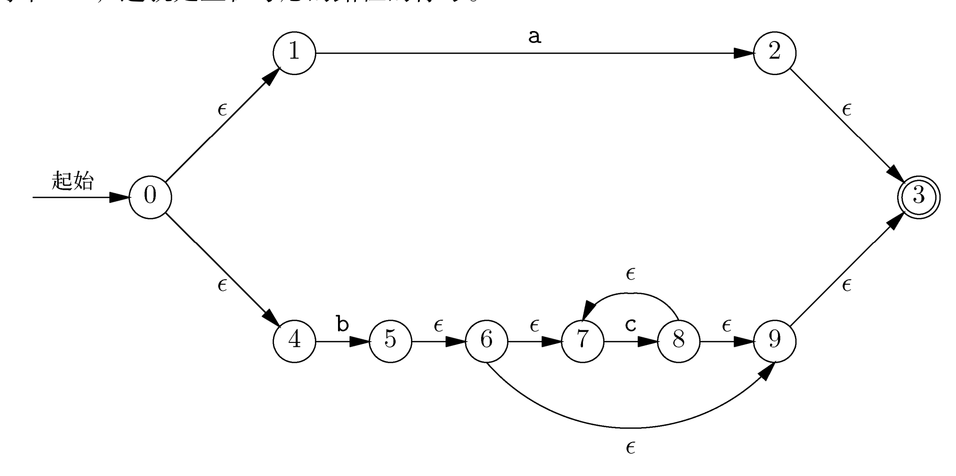


图10-26　表示a|bc*的具有 ϵ 转换的自动机

大家可能会发现，从状态0到状态3的路径的标记只会是a、b、bc、bcc、bccc，等等。表示该集合的正则表达式是**a | bc***，而且我们会看到图10-26中的自动机可以自然地由这一正则表达式构建而来。

10.8.2　从正则表达式到具有 ϵ 转换的自动机

我们可以利用根据对正则表达式中运算符数量进行完全归纳得出的算法，把正则表达式转换成自动机。这一思路类似于我们在5.5节中介绍过的对树的结构归纳，而且如果用正则表达式的表达式树（其中原子操作数是树叶，而运算符在中间节点位置）来表示它们，这种对应就会变得更加明朗。要证明的命题如下。

命题 $S(n)$。如果R是含n个运算符，而且不含变量作为原子操作数的正则表达式，那么存在具有ϵ转换的自动机A，只接受$L(R)$中的字符串。此外，A满足如下所有条件：

(1) 只有一个接受状态；

(2) 没有弧通向它的起始状态；

(3) 没有弧从它的接受状态出发。

依据。如果$n=0$，那么R一定是一个原子操作数，它可能是$\varnothing$、ϵ，或是对应某个符号x的**x**。在这3种情况下，可以设计满足命题$S(0)$要求的双状态的自动机，这些自动机如图10-27所示。

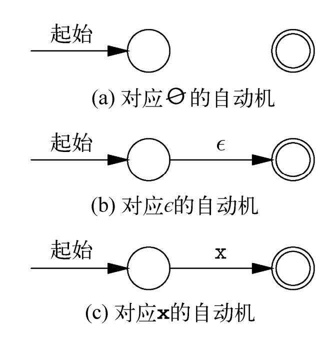


图10-27 依据情况中的自动机

请务必理解，这里为正则表达式中出现的每个操作数创建了新的自动机，而且所具有的状态与其他任何自动机的状态都不一样。例如，如果在表达式中出现3个**a**，我们就会创建3个不同的自动机，总共有6个状态，每个自动机都与图10-27c所示的自动机类似，只不过用**a**替代了其中的**x**。

图10-27a中的自动机显然不接受任何字符串，因为我们没办法从起始状态到达接受状态，因此它的语言为$\varnothing$。图10-27b是对应ϵ的，因为它只接受空字符串。图10-27c是只接受字符串**x**的自动机。我们可以用为符号**x**选择的不同值创建新的自动机。请注意，这些自动机都能满足上述3个要求，只有一个接受状态，没有进入起始状态的弧，也没有从接受状态出发的弧。

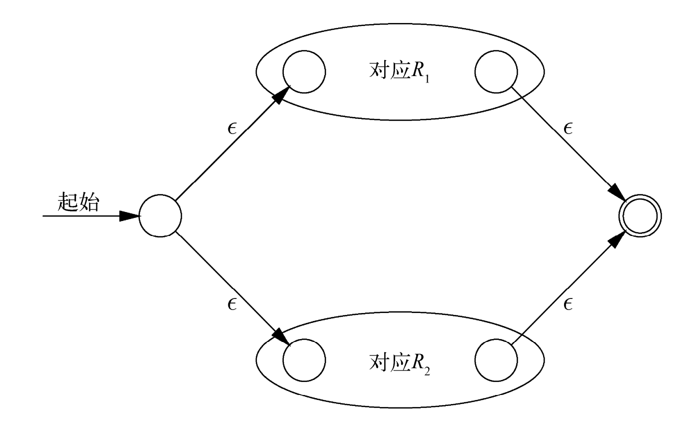
归纳。现在假设$S(i)$对所有的$i \leqslant n$都成立，也就是说，对任意最多具有n个运算符的正则表达式R来说，存在自动机满足归纳假设的条件，并且只接受$L(R)$中的所有字符串。现在，设R是具有n+1个运算符的正则表达式。我们可以将注意力放在R“最外侧”的运算符上，也就是说，R只可能是$R_1 | R_2$、R_1R_2或R_1*这样的形式，具体取决于形成R时最后用到的那个运算符是区别、串接还是闭包。

在这3种情况的任意一种中，R_1和R_2都不可能具有n个以上运算符，因为R中有一个运算符不属于R_1和R_2中的任何一个。[①]因此，归纳假设在所有这3种情况下都是适用于R_1和R_2的。我们可以通过依次考虑这几种情况来证明$S(n+1)$。

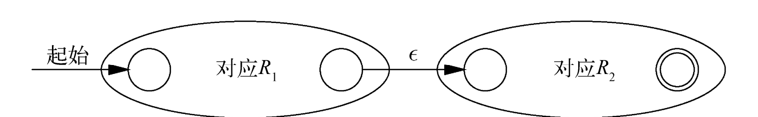
情况1。如果$R = R_1 | R_2$，那么可构建图10-28a中的自动机。取R_1和R_2，并添加两个新状态（一个起始状态和一个接受状态），从而构造了该自动机。与R对应的自动机的起始状态具有到与R_1和R_2对应的自动机起始状态的ϵ转换。这两个自动机的接受状态分别有到对应的自动机接受状态的ϵ转换。不过，与R_1和R_2对应的自动机的起始状态与接受状态不是构造出的自动机的起始状态和接受状态。

① 不要忘了，即便串接是通过并置操作数来表示的，并没有可见的运算符符号，但在确定R中出现了多少个运算符时，仍然要将串接的使用记入运算符出现的次数中。

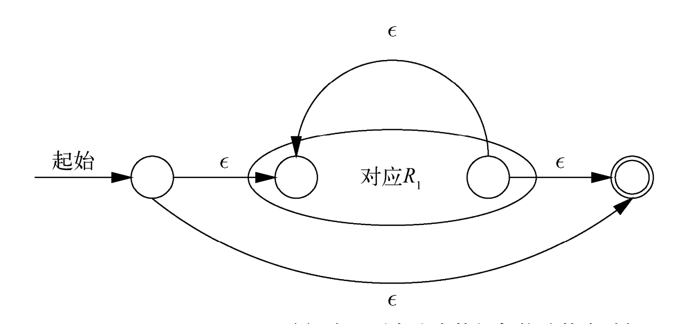
这种构造之所以行得通，是因为从对应R的自动机的起始状态到接受状态的唯一方式，是沿着一条标记为ϵ的弧到与R_1对应或与R_2对应的自动机的起始状态。然后必须沿着所选自动机中的路径到达其接受状态，之后经过ϵ转换到达与R对应的自动机的接受状态。这一路径是由我们行经的自动机所接收的某个字符串s标记的，因为我们从该自动机的起始状态行至了接受状态。因此，s要么在$L(R_1)$中，要么在$L(R_2)$中，这取决于我们行经的自动机到底是哪个。因为我们只为路径的标号增加了ϵ，所以图10-28a中的自动机也接受s。因此被接受的字符串都在$L(R_1)\cup L(R_2)$中，也就是在$L(R_1 | R_2)$，或者说$L(R)$中。



(a) 为两个正则表达式的取并构造的自动机



(b) 为两个正则表达式的串接构造的自动机



(c) 为正则表达式的闭包构造的自动机

图10-28　根据正则表达式构造自动机的归纳部分

情况2。如果$R = R_1R_2$，那么可以构造如图10-28b所示的自动机。该自动机的起始状态是与R_1对应的自动机的起始状态。而它的接受状态是与R_2对应的自动机的接受状态。我们添加了从与R_1对应的自动机的接受状态到与R_2对应的自动机的起始状态的ϵ转换。第一个自动机的接受状态不再是接受状态，而第二个自动机的起始状态在构造的自动机中也不再是起始状态。

在图10-28b所示的自动机中，从起始状态到接受状态的唯一方式如下：

(1) 顺着由$L(R_1)$中某字符串s标记的路径，从起始状态到达与R_1对应的自动机的接受状态；

(2) 接着沿着标记为ϵ的路径到达与R_2对应的自动机的起始状态；

(3) 然后顺着由$L(R_2)$中某字符串t标记的路径，到达其接受状态。

这条路径的标号是st。因此图10-28b中的自动机接受的刚好是$L(R_1 R_1)$，也就是$L(R)$中的字符串。

情况3。如果$R=R_1*$，则可以构造如图10-28c所示的自动机。我们为与R_1对应的自动机添加了新的起始状态和接受状态。这个新的起始状态具有到新接受状态的ϵ转换（所以字符串ϵ会被接受），而且有到与R_1对应的自动机的起始状态的ϵ转换。与R_1对应的自动机的接受状态被赋予了回到其起始状态的ϵ转换，以及到与R对应的自动机的接受状态的ϵ转换。与R_1对应的自动机的起始状态与接受状态不再是构造出的自动机的起始状态与接受状态。

图10-28c中从起始状态到接受状态的路径要么标记为ϵ（如果是直接到达），要么是由$L(R_1)$中一个或多个字符串的串接来标记，一如我们行经与R_1对应的自动机，并且按自己喜好反复回到其起始状态一样。请注意，我们每次在行经与R_1对应的自动机时，并不一定都要沿着相同的路径。因此，经过图10-28c的路径的标号刚好是$L(R_1*)$，也就是$L(R)$中的字符串。

✦ 示例 10.25

下面我们来为正则表达式**a | bc***构造自动机。对应该正则表达式的表达式树如图10-29所示，它类似于我们在5.2节中讨论过的表达式树，并有助于我们了解运算符应用到操作数上的次序。

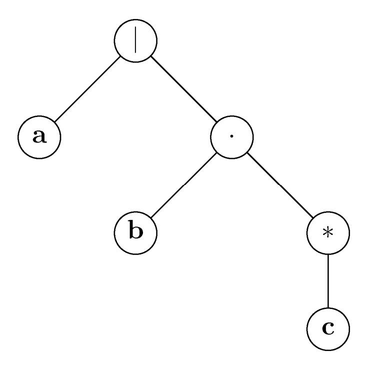


图10-29 与正则表达式**a|bc***对应的表达式树

总共有3个叶子节点，而且我们为每个叶子节点都构造了类似图10-27c所示的自动机实例。这些自动机如图10-30所示，而且使用了与图10-26所示的自动机（正如我们提到过的，这是我们最终要为该正则表达式构造的自动机）一致的状态。不过，大家应该明白，对应多次出现的操作数的自动机有着不同的状态。在这个例子中，因为每个操作数都是不同的，我们能想到要为每个操作数使用不同状态，不过，打个比方说，如果表达式中出现了多个**a**，就要为每个**a**创建不同的自动机。

现在必须应用运算符并构建随着进程更大的自动机，逐步建立起图10-29中的树。最先应用的运算符是闭包运算符，它是应用到操作数**c**上的。我们利用了图10-28c中对应闭包运算的构造方法。引入的新状态分别称为状态6和状态9，还是与图10-26保持一致。图10-31展示了与正则表达式**c***对应的自动机。

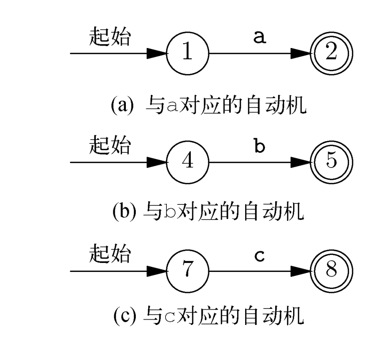


图10-30　对应**a**、**b**和**c**的自动机

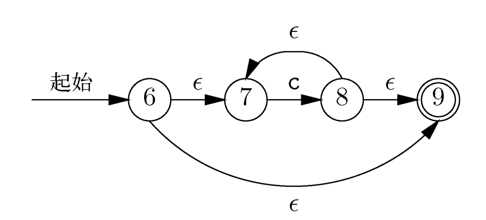


图10-31　对应**c***的自动机

接下来对**b**和**c***应用了串接运算符。利用的是图10-28b的构造方法，得到的自动机如图10-32所示。

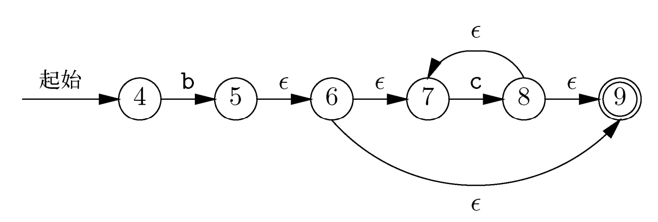


图10-32　对应**bc***的自动机

最后，我们对**a**和**bc***应用取并运算符。这里用到了图10-28a所示的构造方法，而且将引入的新状态称为状态0和状态3，得到的自动机就如图10-26所示。

10.8.3　消除ϵ转换

如果我们在具有ϵ转换的某自动机的任意状态s中，其实也是在从状态s沿着由标记为ϵ的弧形成的路径可以到达的任意状态。原因在于，不管是什么字符串标记了到达状态s所经过的路径，同样的字符串都是用ϵ转换扩展过的该路径的标号。

✦　示例 10.26

在图10-26中，可以沿着标记了b的路径到达状态5。从状态5起，可以沿着由标记了ϵ的弧形成的路径到达状态6、状态7、状态9和状态3。因此，如果我们在状态5中，其实也就在其他4个状态中。例如，因为状态3是接受状态，所以也可以把状态5视作接受状态，因为能把我们带到状态5的每个输入字符串，也能把我们带到状态3，因此是可以被接受的。

因此，要问的第一个问题是，从各状态开始，只沿着ϵ转换可以到达哪些其他状态？在9.7节中，我们在了解深度优先搜索的一种应用（可达性问题）时，给出了回答这一问题的算法。

对这里的问题来说，要对表示有限自动机的图进行的修改，只是要把图中所有除ϵ转换之外的转换删除。也就是说，对每个实际的符号x，要删除所有标记为x的弧。然后从剩下的图中各个节点开始进行深度优先搜索。在从节点v开始的深度优先搜索期间访问过的节点，刚好就是从v开始只使用ϵ转换便可到达的节点组成的集合。

回想一下，深度优先搜索要花费$O(m)$的时间，其中m是图中节点数和弧数的较大者。在这种情况下，如果图中有n个节点，要进行n次深度优先搜索，总共要花$O(mn)$的时间。不过，在通过在本节前面的内容中描述的算法从正则表达式构造的自动机中，任意节点出发的弧最多只有两条。因此$m \leqslant 2n$，而且$O(mn)$就是$O(n^2)$的时间。

✦　示例 10.27

在图10-33中，可以看到从图10-26中删除标记了由实际符号a、b、c标记的3条弧后剩下的弧。图10-34中的表给出了图10-33的可达性信息，也就是说第i行和第j列的1就表示存在从节点i到节点j的长度为0或以上的路径

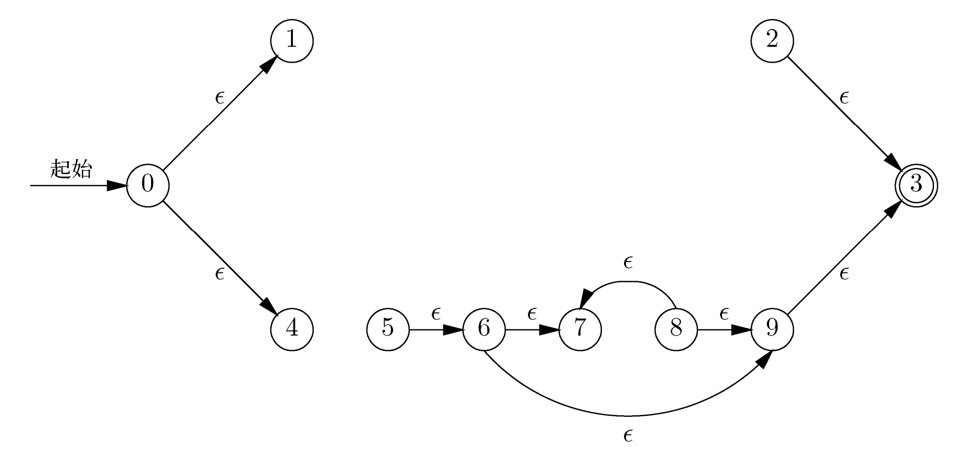


图10-33　图10-26中的ϵ转换

	0	1	2	3	4	5	6	7	8	9
0	1	1			1					
1		1								
2			1	1						
3				1						
4					1					
5				1		1	1	1		1
6				1			1	1		1
7								1		
8				1				1	1	1
9				1						1

图10-34　图10-33对应的可达性表

有了可达性信息之后，就可以构造不含ϵ转换的等价自动机。思路就是把旧自动机中具有0次或多次ϵ转换一条路径以及后面标记为实际符号的一次转换，捆绑成新自动机中的一次转换。

每一次这样的转换，都会将我们带到图10-27c的依据规则（操作数为实际符号时的规则）引入的自动机中第二个状态。原因在于，这些状态只有以真实符号为标号的弧才进入。因此，我们的新自动机只需要这些状态，以及对应其自身状态集的起始状态。可以把这些状态叫作*重要状态*。

在构造的新自动机中，如果存在某个状态k满足以下条件，就有从重要状态i到重要状态j的，标号中含有符号x的转换。

(1) 从状态i沿着具有0次或多次ϵ转换的路径可达状态k。请注意，$k=i$一直是可以的。

(2) 在旧自动机中，存在从状态k到状态j，标记为x的转换。

我们还必须决定新自动机中哪些状态是接受状态。正如上文提到过的，当我们在某个状态中时，其实就是在由该状态沿着标记为ϵ的弧所能到达的任意状态中。因此如果在旧自动机中从状态i到其接受状态之间存在由标记为ϵ的弧形成的路径，就将状态i作为新自动机的接受状态。请注意，状态i本身可能就是旧自动机的接受状态，并因此在新自动机中仍然是接受状态。

✦ 示例 10.28

我们来将图10-26所示的自动机转变为接受相同语言但不带ϵ转换的自动机。首先，重要状态包括作为初始状态的状态0，以及由标记了实际符号的弧进入的状态2、状态5和状态8。

首先从找到状态0对应的转换开始。根据图10-34，从状态0可以沿着由标记了ϵ的弧构成的路径到达状态0、状态1和状态4。我们发现有针对a的从状态1到状态2的转换，而且有针对b的从状态4到状态5的转换。因此，在新自动机中，存在从状态0到状态2的标记为a的转换，并存在从状态0到状态5的标记为b的转换。请注意，我们已经把图10-26中的路径$0\to1\to2$和$0\to4\to5$压缩成标记了这些路径上非ϵ转换的标号的转换。因为状态0，以及从它出发沿着以ϵ标记的路径可达的状态1和状态4都不是接受状态，所以在新自动机中，状态0不是接受状态。

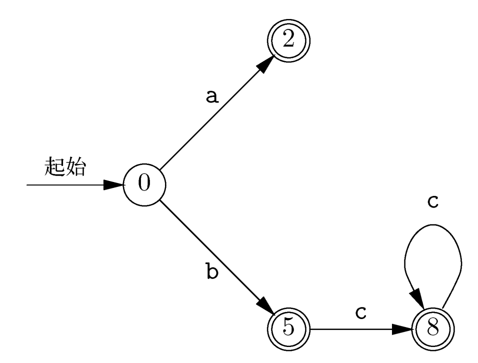


图10-35　通过消除ϵ转换，根据图10-26构造的自动机。请注意，该自动机只接受L(a|bc*)中的所有字符串

接下来，考虑从状态2出发的转换。图10-34告诉我们，从状态2出发，通过ϵ转换只能到达它本身和状态3，因此我们必须寻找从状态2或状态3出发针对实际符号的转换。因为没找到，所以可知在新自动机中没有从状态2出发的转换。不过，状态3是接受状态，而且从状态2经过ϵ转换可以到达状态3，所以状态2就是新自动机的接受状态。

在考虑状态5时，图10-34表明要查看状态3、状态5、状态6、状态7和状态9。在这些状态中，只有状态7具有从自身出发的非ϵ转换，它被标记为c，而且通向状态8。因此，在新自动机中，从状态5出发的唯一转换就是针对c的到状态8的转换。因为从状态5沿着标记了ϵ的弧可以到达接受状态3，所以我们把状态5也作为新自动机的接受状态。

最后，一定要查看从状态8出发的转换。经过类似对状态5进行的推理可以得出，在新自动机中，从状态8出发的唯一转换是到它自身的，而且被标记为c。因此，状态8也是新自动机的接受状态。

图10-35展示了该新自动机。请注意，它接受的字符串集正好是*L*(**a**|**bc***)中的那些字符串，也就是将我们带到状态2的字符串a，将我们带到状态5的字符串b，以及bc、bcc、bccc这些将我们带到状态8的一系列字符串。图10-35中的自动机刚好是确定自动机。如果它不是，假设想设计可以识别原正则表达式的字符串的程序，就可以利用子集构造将其转化为确定自动机。

顺便提一下，存在与图10-35接受同一语言的更加简化的确定自动机，该自动机如图10-36所示。事实上，只要意识到状态5和状态8是等价并可以被合并的，就可以得到这一改进过的自动机。得到的状态就是图10-36中的状态5。

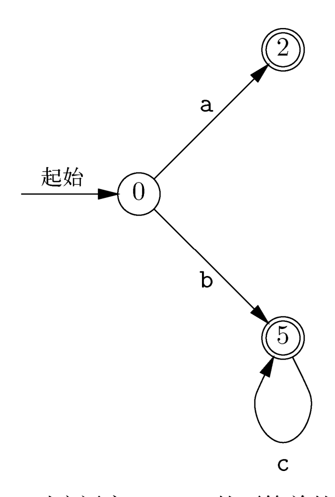


图10-36 对应语言*L*(**a**|**bc***)的更简单的自动机

10.8.4 习题

(1) 为以下正则表达式构造具有 ϵ 转换的自动机。

(a) **aaa**。提示：请记住，要为出现的每个**a**创建新自动机。

(b) **(ab|ac)***。

(c) **(0|1|1*)***。

(2) 为习题(1)中构造的各个自动机找到由标记为 ϵ 的弧构成的图中节点的可达集。请注意，在大家构造不含 ϵ 转换的自动机时，只需要为初始状态和那些有非 ϵ 转换进入的状态构造可达状态。

(3) 为习题(1)中构造的各个自动机构造不含 ϵ 转换的等价自动机。

(4) 习题(3)得出的自动机中哪些是确定自动机？为其中那些非确定自动机构造等价的确定自动机。

(5) * 对由习题(3)和习题(4)构造的确定自动机而言，是否存在状态更少的等价确定自动机？如果有，找出状态最少的那个。

(6) * 我们可以扩展从正则表达式构造含 ϵ 转换的自动机的过程，将正则表达式的范围扩大到包含那些使用了10.7节中扩展过的运算符的表达式。这一命题从原则上讲是成立的，因为那些扩展都是“原始”正则表达式的简略形式，我们只是用扩展的运算符替代了原有的表达式而已。不过，还可以直接把扩展过的运算符融入到我们的构造过程中。说明如何修改构造过程以涵盖下列运算符：

(a) ? 运算符（不出现或出现1次）；

(b) $^+$运算符（出现1次或更多次）；

(c) 字符类。

(7) 我们可以修改把正则表达式转化为自动机的算法中对应串接的情况。在图10-28b中，引入了从与 R_1 对应自动机的接受状态到与 R_2 对应自动机的初始状态的 ϵ 转换。另一种方式就是按照图10-37所示的方式合并 R_1 的接受状态到 R_2 的初始状态。使用旧算法与修改过的算法为正则表达式**ab*c**构造自动机。

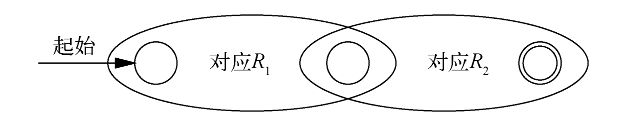


图10-37　另一种与两个正则表达式的串接对应的自动机

10.9　从自动机到正则表达式

本节中还要展示自动机与正则表达式等价性的另一半，证实对每个自动机*A*，都存在语言刚好是*A*所接受的字符串集合的正则表达式。尽管我们一般会使用10.8节中的构造过程，其中要把正则表达式形式的“设计”转化为确定自动机形式的程序，不过把自动机转化为正则表达式的构造过程也是很有趣很有益的,它完成了这两种截然不同的模式表示法的表现力之间的等价性的证明。

我们的构造过程涉及从自动机中一个一个地删除状态。随着构造过程的进行，会把弧上的标号由最初的字符串集合替换为更复杂的正则表达式。一开始，如果弧上的标号是{ $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, …, $\boldsymbol{x}_n$ }，就可以把这些标号替换为正则表达式 $\mathbf{x}_1 | \mathbf{x}_2 | \cdots | \mathbf{x}_n$ ，该正则表达式从本质上讲表示的是相同的符号集合，虽然严格地讲正则表达式表示的是长度为1的字符串。

一般而言，可以将路径的标号视为路径沿线上正则表达式的串接，或是看作这些表达式的串接定义的语言。这一观点与我们用字符串标记路径的概念是一致的。也就是说，如果路径的弧是用正则表达式 R_1 、R_2 、…、R_n 按此次序标记的，则当且仅当字符串w在语言$L(R_1 R_2 \cdots R_n)$中时有该路径被标记为w。

✦　示例 10.29

考虑图10-38中的路径 $0 \to 1 \to 2$ 。正则表达式**a|b**和**a|b|c**依次标记了这两条弧，因此标记该路径的字符串集合是由正则表达式(**a** | **b**)(**a** | **b** | **c**)定义的语言中的字符串构成的，也就是{aa，ab，ac，ba，bb，bc}。

图10-38　以正则表达式作为标号的路径，路径的标号是由正则表达式的串接定义的语言

10.9.1　状态消除的构造

在从自动机到正则表达式的转化中，关键的步骤就是状态的消除，如图10-39所示。我们希望消除状态*u*，不过必须保留弧的正则表达式标号，从而使剩余状态中两两之间路径的标号集合

不发生改变。在图10-39中，状态u的前导分别是s_1、s_2、…、s_n，而u的后继则分别是t_1、t_2、…、t_m。虽然已经证明了这些s和t是不相交的状态集，但其实两组中还是可能存在一些相同的状态。

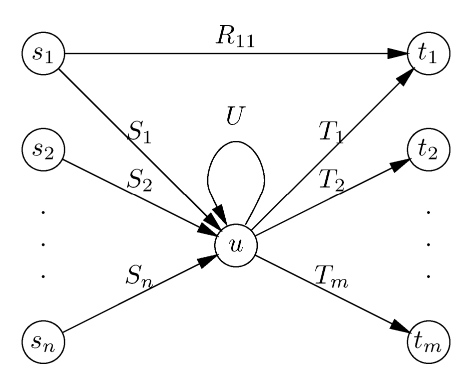


图10-39　我们想消除状态u

不过，如果u是它本身的后继，我们就要用标记为U的弧明确表示这一事实。假设在状态u处没有这样的自环，那么可以引入一个这样的自环，并赋予其标号$\varnothing$。标号为$\varnothing$的弧是“不存在的”，因为任意用到这条弧的路径标号都会是含有$\varnothing$的正则表达式的串接。因为$\varnothing$是串接的零元，所以这样的串接定义的都是空语言。

我们还要明确给出从s_1到t_1的弧R_{11}。一般而言，假设对每个i=1、2、…、n，以及对每个j=1、2、…、m，都存在从s_i到t_j的弧，由某个正则表达式R_{ij}标记。如果弧$s_i \rightarrow t_j$实际上不存在，就引入它并为其赋予标号$\varnothing$。

最后，在图10-39中存在从各状态s_i到u的，由正则表达式s_i标记的弧，而且存在从u到各状态t_j的，由正则表达式T_j标记的弧。如果消除节点u，那么这些弧与图10-39中标记为U的弧都将不复存在。要让标记路径的字符串集合保持不变，就必须考虑每对s_i和t_j，并为弧$s_i \rightarrow t_j$的标号添加一个能表示所失去内容的正则表达式。

在消除u之前，标记了从s_i到u（包括多次行经的那些$u \rightarrow u$自环）然后从u到t_j的路径的那些字符串集合是由正则表达式$S_iU^*T_j$描述的。也就是说，$L(S_i)$中的字符串可以把我们从状态s_i带到到状态u，$L(U^*)$中的字符串可以把我们从状态u带到状态u，沿着该自环0次、1次或更多次。最后，$L(T_j)$中的字符串把我们从状态u带到状态t_j。

因此，在消除状态u和所有进出u的弧之后，必须把弧$s_i \rightarrow t_J$的标号由R_{ij}替换为$R_{ij} \mid S_iU^*T_j$。

存在不少实用的特例。首先，若U=$\varnothing$，即u上的自环并非真正存在，那么$U^* = \varnothing^* = \epsilon$。因为$\epsilon$是串接的单位元，所以$(S_i\epsilon)T_j = S_iT_j$，也就是说，$U$其实在它应该出现的位置消失了。同样，如果$R_{ij} = \varnothing$，意味着之前没有从$s_i$到$t_j$的弧，我们就引入这条弧，并给予其标号$S_iU^*T_j$，或者如果$U = \varnothing$，就是$S_iT_j$。这样做的原因在于，$\varnothing$是取并运算的单位元，因此$\varnothing \mid S_iU^*T_j = S_iU^*T_j$。

✦　示例 10.30

我们来考虑一下图10-4所示的反弹过滤器自动机，这里的图10-40重现了该自动机。假设要消除状态b，它们就扮演了图10-39中u的角色。状态b有一个前导a，以及两个后继a和c。b上不存在自环，所以要引入一个标号为$\varnothing$的自环。存在从a到其本身，标记为**0**的弧。因为a既是b的前导又是b的后继，所以该弧在这一变形中是必须的。唯一一对另外的前导–后继对是a和c。因为不存在

弧 $a \to c$ ，所以可以添加一条标号为 $\varnothing$ 的弧 $a \to c$ 。相关状态和弧组成的图如图10-41所示。

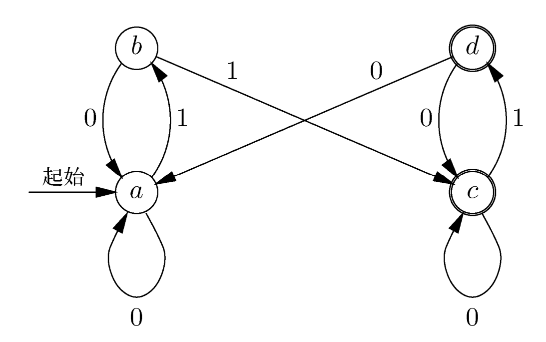


图10-40　与反弹过滤器对应的有限自动机

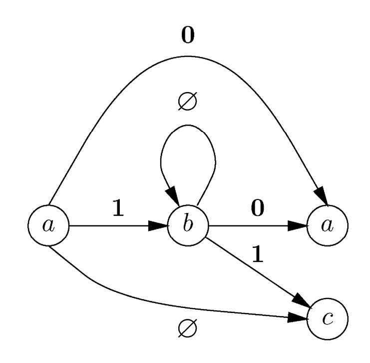


图10-41　状态*b*，以及它的前导和后继

对状态对*a*-*a*而言，我们把弧 $a \to a$ 的标号替换为 $0|1\varnothing{*}0$ 。`0`这项表示该弧的原始标号，而`1`这项是 $a \to b$ 的标号，$\varnothing$ 是自环 $b \to b$ 的标号，而第二个`0`项则是弧 $b \to a$ 的标号。我们可以按照之前的描述进行简化，消除 $\varnothing{*}$，留下表达式`0|10`，这是说得通的。在图10-40中，从*a*到*a*的路径，行经*b*状态0次或多次，而不经过其他状态，其标号集合为{0，10}。

处理状态对*a*-*c*的过程是类似的。我们要用可以简化为`11`的 $\varnothing|1\varnothing{*}1$ 替代弧 $a \to c$ 的标号 $\varnothing$ 。这还是说得通的，因为在图10-40中，从*a*到*c*的唯一路径，经过*b*而且标号为11。在消除节点*b*并改变弧标号后，图10-40就成了图10-42。请注意，在该自动机中，某些弧标号中的正则表达式具有长度大于1的字符串。不过，状态*a*、*c*和*d*之间的路径对应的路径标号集合与图10-40相比没有发生改变。

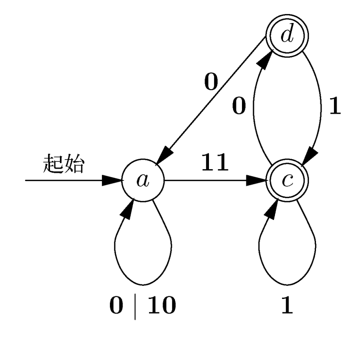


图10-42　消除了状态*b*之后的反弹过滤器自动机

10.9.2 自动机的完全简化

要得到只表示所有由自动机A接受的字符串的正则表达式，就要依次考虑A的各接受状态t。每个被A接受的字符串之所以会被接受，是因为它标记了从起始状态s到某个接受状态t的路径。可以按照以下方式，为那些把我们从s带到某个特定接受状态t的字符串构造相应的正则表达式。

反复消除自动机A的状态，直到只剩s和t两个状态，这样一来，该自动机就如图10-43这样了。我们已经展示了所有4条可能的弧，每一条都以一个正则表达式作为其标号。如果一条或多条可能的弧不存在，可以引入该弧并为其标记上$\varnothing$。

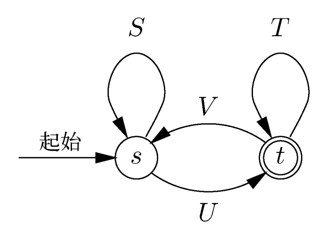


图10-43 减少到两个状态的自动机

需要找出有哪些正则表达式描述了始于s并终于t的路径的标号集合。表示该字符串集合的一种方式是认识到每一条这样的路径会先到达t，然后从t行至其自身0次或多次，还可能在行进的过程中经过s。一开始把我们带到状态t的字符串集合是$L(S^*U)$。也就是说，要用到$L(S)$中的字符串0次或多次，这样做就会先留在状态s中，然后沿着$L(U)$中的字符串行进。我们既可以跟随$L(T)$中的字符串停留在状态t中，这会将我们从t带到t，也可以沿着VS^*U中的字符串到达s，在s停顿一会儿，然后又回到t。我们可以沿着这两组中以任意次序排列的0个或多个字符串行进，并将其表示为$(T\,|\,VS^*U)^*$。因此从状态s到状态t的字符串集合对应的正则表达式就是

$$S^*U(T\,|\,VS^*U)^* \tag{10.4}$$

存在一种特例，就是起始状态s本身也是接受状态的情况。这样的话，有些字符串被接受的原因是因为它们将自动机A从状态s带到了状态s。我们消除了除s之外的所有状态，留下如图10-44所示的自动机。将A从状态s带到状态s的字符串集合是$L(S^*)$。因此可以用S^*作为取代接受状态s的贡献的正则表达式。

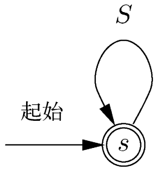


图10-44 只有起始状态的自动机

将起始状态为s的自动机A转化成等价正则表达式的完整算法如下所述。对每个接受状态t而言，从自动机A开始，并消除各种状态，直到只剩下状态s和t。当然，对每个接受状态t来说，都要从全新的原自动机A开始进行处理。

如果$s \neq t$，就使用(10.4)式得出其语言是把A从状态s带到状态t的字符串集合的正则表达式。如果$s = t$，就利用S^*，其中S是弧$s \to s$的标号。然后，为对应每个接受状态t的正则表达式取并。该表达式的语言就刚好是被A接受的字符串的集合。

✦ 示例 10.31

下面来为图10-40所示的反弹过滤器自动机得出相应的正则表达式。因为c和d是接受状态，所以需要进行下列操作：

(1) 从图10-40中消除状态b和d，得到只涉及a和c的自动机；

(2) 从图10-40中消除状态b和c，得到只涉及a和d的自动机。

因为在这两种情况下都必须消除状态b，所以图10-42就让我们的最终目标实现了一半。对情况(1)，要在图10-42的基础上消除状态d。存在从c经过d到a的标号为`00`的路径，所以需要引入一条从c到a标记为`00`的弧。存在从c经过d回到其自身的标号为`01`的路径，因此需要为c处的自环添加标号`01`，这样该标号就成了`1|01`。得到的自动机如图10-45所示。

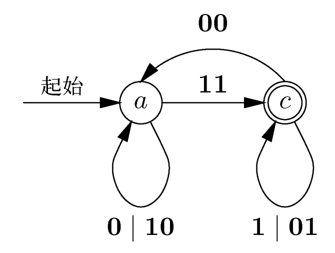


图10-45　把图10-40所示的自动机减少到只剩状态a和状态c

对目标(2)，要再次从图10-42开始，而这次要消除状态c。在图10-42中，我们可以从a经过c到达d，而描述可能字符串的正则表达式为`111*0`。[①]也就是说，`11`将我们从a带到c，`1*`让我们在c处循环0次或多次，而最后`0`把我们从c带到d。因此，我们引入了从a到d的标号为`111*0`的弧。同样，在图10-42中，可以沿着`11*0`中的字符串，从d通过c行至其自身。因此，这一表达式成了d处自环的标号。简化过的自动机如图10-46所示。

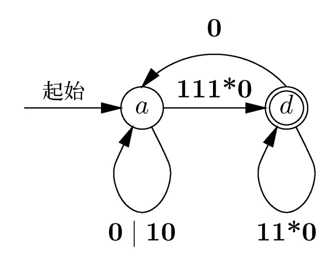


图10-46　把图10-40中的自动机减少到只剩状态a和状态d

现在可以把(10.4)中得出的公式应用到图10-45和图10-46所示的自动机上。对图10-45，有$S=0|10$，$U=11$，$V=00$，而且$T=1|01$。因此表示将图10-40所示的自动机从起始状态a带到接受状态c的字符串集合的正则表达式为

$$\texttt{(0|10)*11}\big(\texttt{(1|01)|00(0|10)*11}\big)\texttt{*} \tag{10.5}$$

而表示把该自动机从起始状态a带到接受状态d的字符串的正则表达式为

$$\texttt{(0|10)*111*0}\big(\texttt{11*0|0(0|10)*111*0}\big)\texttt{*} \tag{10.6}$$

① 请记住，因为*的优先级高于串接，`111*0`会被解释为`11(1*)0`，并表示由两个或更多1后面加以一个0构成的字符串。

表示由反弹过滤器自动机接受的字符串的正则表达式就是对(10.5)和(10.6)取并，或者说是

((0|10)*11((1|01)|00(0|10)*11)*)| ((0|10)*111*0(11*0|0(0|10)*111*0)*)

没有办法对该表达式进行多少简化，因为相同的因式只有(0|10)*11，其他就基本没有什么相同的了。我们可以删除(10.5)中因式(1|01)周围的括号，因为取并运算是具有结合性的，这样得到的表达式就是

(0|10)*11((1|01|00(0|10)*11)*)|1*0(11*0|0(0|10)*111*0)*)

大家可以回想一下，我们为同样的语言提出过一个简单得多的正则表达式

(0|1)*11(1|01)*(ϵ |0)

这一区别应该提醒我们，对同一语言来说，可能存在不止一个与之对应的正则表达式，而通过转化自动机得到的正则表达式也不一定是对应该语言的最简表达式。

10.9.3 习题

(1) 分别找出下列各图所示自动机对应的正则表达式。

(a) 图10-3

(b) 图10-9

(c) 图10-10

(d) 图10-12

(e) 图10-13

(f) 图10-17

(g) 图10-20

大家可能会希望利用10.6节中的简略形式。

(2) 把10.4节习题(1)中的自动机转化成正则表达式。

(3) * 证明，把我们从图10-43中的状态s带到状态t的字符串集合对应的另一个正则表达式是$(S|UT^*V)^*UT^*$。

(4) 如何修改本节中的构造过程，使得正则表达式可以由具有ϵ转换的自动机生成？

10.10 小结

10.4节的子集构造，以及10.8节和10.9节中的转化，告诉我们3种表示语言的方式有着相同的表现效用。也就是说，针对某语言L的以下3个命题要么都为真，要么都为假。

(1) 存在某确定自动机只接受L中的所有字符串。

(2) 存在某（可能为非确定的）自动机只接受L中的所有字符串。

(3) L对某正则表达式R来说是$L(R)$。

子集构造说明了(2)可以得到(1)。显然有(1)就有(2)，因为确定自动机就是一种特殊形式的非确定自动机。我们在10.8节中证明了由(3)可以得到(2)，而在10.9节中证实了有(2)就有(3)。因此，(1)、(2)和(3)是等价的。

除了这些等价关系外，我们还应该从第10章获得一些重要思路。

- 确定自动机可作为识别字符串多种不同模式的程序的核心。
- 正则表达式通常是一种用于描述模式的便利表示方法。
- 正则表达式的代数法则使得取并和串接与加法和乘法有着类似的性质，不过还是存在一些区别。

10.11　参考文献

读者在Hopcroft and Ullman [1979]中可以了解更多与自动机和语言理论有关的内容。

虽然有很多类似的模型更早或是同时被提出，但处理字符串的自动机模型最早是由Huffman [1954]中描述的形式粗略表示的，这段历史可以在Hopcroft and Ullman [1979]中找到。正则表达式以及它们与自动机的等价性源自Kleene [1956]。非确定自动机以及子集构造来自Rabin and Scott [1959]。10.8节利用的从正则表达式构造非确定自动机的过程源自McNaughton and Yamada[1960]，而10.9节中相反方向的构造过程则来自Kleene的论文。

用正则表达式描述字符串中的模式最早见于Ken Thompson的QED系统（Thompson [1968]），同样的思路随后影响到他开发的UNIX系统中的很多命令。正则表达式在系统软件中还有很多其他的应用，Aho, Sethi and Ullman [1986]描述了其中大量应用。

Aho, A. V., R. Sethi , and J. D. Ullman [1986]. *Compiler Design: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass.

Hopcroft, J.E. and J. D. Ullman [1979]. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass.

Huffman, D. A. [1954]. “The synthesis of sequential switching machines,” *Journal of the Franklin Institute* **257**:3-4, pp. 161–190 and 275–303.

Kleene, S. C. [1956]. “Representation of events in nerve nets and finite automata,” in *Automata Studies* (C. E. Shannon and J. McCarthy , eds), Princeton University Press.

McNaughton, R. and H. Yamada [1960]. “Regular expressions and state graphs for automata,” *IEEE Trans. on Computers* **9**:1, pp. 39–47.

Rabin, M. O. and D. Scott [1959]. “Finite automata and their decision problems,” *IBM J. Research and Development* **3**:2, pp. 115-125.

Thompson, K. [1968]. “Regular expression search algorithm,” *Comm. ACM* **11**:6, pp. 419–422.

第 11 章

模式的递归描述

在第10章中，我们看到过两种等价的模式描述方式。一种是图论方式，利用了一种名为“自动机”的图中路径的标号。另一种是代数方式，利用了正则表达式。在本章中，我们将看到第三种描述模式的方式，利用到了一种名为“上下文无关文法”（以下简称“文法”）的递归定义。

文法的重要应用之一就是作为编程语言的规范。文法是用来描述常见编程语言句法的一种简洁表示方式，我们会在本章中看到很多示例。此外，有一种机械的方式可以把常见编程语言的文法转换成“分析器”（parser）——该语言编译器的一个关键部分。分析程序揭示了源程序的结构，通常是将程序中的每条语句表示为表达式树的形式。

11.1 本章主要内容

本章主要讨论如下主题。

- 文法以及文法是如何用来定义语言的（11.2节和11.3节）。
- 分析树，根据给定文法显示字符串结构的树表示（11.4节）。
- 歧义，当某一字符串有两棵或更多分析树，并因此根据给定文法不具有唯一“结构”时出现的问题（11.5节）。
- 把文法转换成“分析器”的一种方法，“分析器”是可以分辨给定字符串是否在某一语言中的算法（11.6节和11.7节）。
- 证明文法在描述语言方面要比正则表达式更强大（11.8节）。首先，我们通过证明如何用文法模拟正则表达式，来证明文法的描述性至少与正则表达式一样强。接着我们将描述一种只能用文法来指定而不能用正则表达式指定的特殊语言。

11.2 上下文无关文法

算术表达式可以由递归定义自然而然地定义出来。下面的示例说明了这一定义是如何起效的。我们来考虑涉及如下内容的算术表达式。

(1) 4种二元运算符+、−、*和/；

(2) 用于分组的圆括号；

(3) 作为操作数的数字。

这种表达式的一般定义是具有如下形式的归纳。

依据。一个数字是一个表达式。

归纳。如果E是表达式，那么以下各种也都是表达式。

(a) (E)。也就是说，我们可以在表达式周围放上圆括号以得到新表达式。

(b) $E+E$。也就是说，用加号连接的两个表达式是一个新表达式。

(c) $E-E$。这一条以及接下来的两条规则都与(b)类似，不过使用了其他的运算符。

(d) $E * E$。

(e) E/E。

这一归纳定义了一种语言，也就是一个字符串集合。依据陈述了任意数字都在该语言中。规则(a)表示，如果s是该语言中的字符串，那么加过括号的字符串(s)也在该语言中，这一字符串是s前面加上左括号并且后面跟上右括号得到的。规则(b)到规则(e)是说，如果s和t是该语言中的两个字符串，那么$s+t$、$s-t$、$s * t$和s/t也都是该语言中的字符串。

文法让我们可以写出这些简明而且含义精确的规则。举例来说，可以用图11-1所示的文法写出我们对算术表达式的定义。

```
(1)  <表达式> → 数字
(2)  <表达式> → (<表达式>)
(3)  <表达式> →<表达式>+<表达式>
(4)  <表达式> →<表达式>–<表达式>
(5)  <表达式> →<表达式>*<表达式>
(6)  <表达式> →<表达式>/<表达式>
```

图11-1　对应简单算术表达式的文法

这里要对图11-1中用到的符号作出一些解释，符号

<表达式>

称为*语法分类*（syntactic category），它代表这一算术表达式语言中的任意字符串。符号→的含义是“可由……组成”。例如，图11-1中的规则(2)就表示，表达式可由左括号后跟上属于表达式的任意字符串再跟上右括号组成。规则(3)表明，表达式可由属于表达式的任意字符串、字符+，以及属于表达式的任意其他字符串组成。规则(4)到规则(6)与规则(3)是相似的。

规则(1)则不同，因为箭头右侧的符号*数字*从字面上看本不是字符串，它只是与可以解释为数字的字符串相对应的占位符。我们在后面的内容中会介绍如何用文法定义数字，但现在先只把*数字*当作一个抽象符号，而表达式用该符号来表示任意原子操作数。

11.2.1　与文法相关的术语

文法中会出现3种符号。第一种是“元符号”，是那些扮演特殊角色而并不代表它们自身的符号。我们目前为止已经见过的元符号只有→，它的用途是把要定义的语法分类与该语法分类中字符串可能的组成方式分隔开。第二种符号是语法分类，我们说过，这种符号表示的是要定义的字符串集合。第三类符号称为终结符（terminal）。终结符可以是+或（这样的字符，也可以是*数字*这样的抽象符号，它代表我们希望在随后定义的字符串。

文法是由*产生式*（production）组成的。图11-1中的每一行都是一个产生式。一般而言，产生式具有以下3个部分。

(1) *左部*（head），就是箭头符号左侧的语法分类。

(2) 元符号→。

(3) *右部*（body），由箭头右侧0个或以上的语法分类和（或）终结符组成。

例如，在图11-1的规则(2)中，左部是<*表达式*>，而右部则由终结符（、语法分类<*表达式*>和终结符）组成。

✦ 示例 11.1

我们可以通过为数字提供定义来扩展本节开始时对表达式的定义。这里要假设数字是由数码（digit）组成的字符串。借用10.6节中扩展过的正则表达式表示法，可以说

digit = [0-9]

number = $digit^+$

不过也可以用文法表示法来表示同样的概念,我们可以写出以下产生式

<*数码*> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<*数字*> → <*数码*>

<*数字*> → <*数字*> <*数码*>

请注意，根据我们对元符号 | 的约定，第一行其实是以下10个产生式的简化形式。

<*数码*> → 0

<*数码*> → 1

…

<*数码*> → 9

可以用同样的方法把对应<*数字*>的两个产生式合并成一行。请注意，对应<*数字*>的第一个产生式表示单个数码是个数字，而第二个产生式的意思是，任何数字后跟上另一个数码也是数字。这两个表达式一起就表示任何由数码组成的串都是数字。

图11-2是扩展过的表示表达式的文法，其中抽象的终结符*数字*被替换为定义了该概念的产生式。请注意，该文法含有3个语法分类<*表达式*>、<*数字*>和<*数码*>。我们会把语法分类<*表达式*>当作起始符号（start symbol），它生成了要用该文法定义的串，在这种情况中，就是格式标准的算术表达式。其他两个语法分类<*数字*>和<*数码*>代表的补充概念是很关键的，但不是写出该文法所需的主要概念。

表示方式的约定

在表示语法分类时，我们是在其斜体名称（中文为楷体）的两侧加上尖括号表示的，例如，<*表达式*>。产生式中的终结符或者用代表字符串x的粗体**x**来表示（类似正则表达式的约定），或者在终结符为抽象符号的情况下（比如之前例子中的*数字*），用不带尖括号斜体字符串（中文为楷体）表示。

元符号 ϵ 表示空右部。因此产生式 $<S> \to \epsilon$ 意味着语法分类$<S>$的语言中包含了空字符串。我们有时会将某一语法分类的右部合并到一个产生式中，分别用可以称作“或”的元符号 | 隔开。例如，如果有以下产生式

$$<S> \to B_1,\ <S> \to B_2,\ \cdots,\ <S> \to B_n$$

其中这些B分别是对应语法分类$<S>$的各产生式的右部，那么可以将这些产生式写为

$$<S> \to B_1 \mid B_2 \mid \cdots \mid B_n$$

---

(1)　<数码> → 0|1|2|3|4|5|6|7|8|9

(2)　<数字> →<数码>
(3)　<数字> →<数字> →<数码>

(4)　<表达式> →<数字>
(5)　<表达式> → （<表达式>）
(6)　<表达式> →<表达式> +<表达式>
(7)　<表达式> →<表达式> −<表达式>
(8)　<表达式> →<表达式> *<表达式>
(9)　<表达式> →<表达式> /<表达式>

图11-2　与由用文法定义的数字组成的表达式对应的文法

✦　示例 11.2

2.6节中曾讨论过平衡圆括号串的概念。当时我们是用非正式的方式给出了组成这种串的归纳定义，而正式形式的书写文法正是本节要介绍的。我们定义了“平衡圆括号串”的语法分类，这里称其为<平衡的>。依据规则陈述了空串是平衡的。可以将该规则写为如下产生式

<平衡的> → ϵ

然后就是归纳步骤，说的是如果x和y都是平衡圆括号串，那么$(x)y$也是。可以把这一规则写为产生式

<平衡的> → （<平衡的>）<平衡的>

因此，图11-3中的文法可以说是定义了平衡圆括号串。

<平衡的> → ϵ
<平衡的> → （<平衡的>）<平衡的>

图11-3　对应平衡圆括号串的文法

还有另一种定义平衡括号串的方式。回想一下2.6节，我们描述这种串的原始动机就是，它们是删除了表达式中除括号外所有内容后留下的括号的子序列。图11-1给出了对应表达式的文法。考虑一下，如果删除掉除括号之外的所有终结符，会发生什么。此时产生式(1)就成了

<表达式> → ϵ

产生式(2)变成了

<表达式> →（<表达式>）

而产生式(3)到产生式(6)都成了

<表达式> →<表达式><表达式>

如果用更为合适的名称<平衡表达式>来代替<表达式>，就得到另一种表示平衡圆括号串的文法，如图11-4所示。这些产生式都是相当自然的。它们表明了如下3点。

(1) 空串是平衡的；

(2) 如果为平衡的串加上圆括号，得到的串也是平衡的；

(3) 而且平衡串的串接是平衡的。

<平衡表达式> → ϵ
<平衡表达式> → （<平衡表达式>）
<平衡表达式> → （<平衡表达式>）<平衡表达式>

图11-4　由算术表达式文法发展来的表示平衡圆括号串的文法

图11-3和图11-4中的文法看起来相当不同，不过它们定义了相同的字符串集合。证明它们确实定义了同一字符串集合最简单的方式也许就是证明由图11-4中<平衡表达式>定义的圆括号串刚好就是2.6节中定义的“量变平衡”圆括号串。好了，现在证明了与图11-3中<平衡的>定义的圆括号串有关的相同断言。

共用文法模式

示例11.1用了两个对应<数字>的产生式来说明“数字是由数码组成的串”。这种模式就是共用的。一般而言，如果有语法分类<*X*>，而且Y是终结符或是另一个语法分类，产生式

$$<X> \rightarrow <X>Y|Y$$

说明任意由*Y*组成的串都是<*X*>。如果用正则表达式来表示，就是$<X> = Y^+$。同样，产生式

$$<X> \rightarrow <X> Y|\epsilon$$

就表示每个由0个或更多的*Y*组成的串都是<*X*>，或者说$<X> = Y^*$。由

$$<X> \rightarrow <X> ZY|Y$$

这样一对产生式表示的也是种共用模式，表示每个开头和结尾都是*Y*而且由*Y*和*Z*交替组成的串都是<*X*>。也就是说，$<X> = Y(ZY)^*$。

此外，我们可以反转上述3个例子任意一个中递归产生式右部中符号的次序，例如

$$<X> \rightarrow Y<X>|Y$$

也定义了$<X> = Y^+$。

✦　示例 11.3

还可以用文法的方式来描述C语言这样的编程语言中控制流的结构。举个简单的例子，想象条件和简单语句这两个抽象终结符。前者代表条件表达式。我们可以把这一终结符替换为语法分类，假如说是<条件>。<条件>的产生式可以用之前所述的表达式文法来构建，但是用到的运算符包含了`&&`这样的逻辑运算符、<这样的比较运算符，以及算术运算符。

终结符简单语句代表不含嵌套控制结构的语句，比如赋值、函数调用、读、写或跳转语句。我们再次用语法分类和扩展它的产生式代替了这一终结符。

这里要用<语句>表示C语言语句的语法分类。形成语句的方式之一是通过`while`结构。也就是说，如果有一条可作为循环体的语句，就可以在它之前加上关键词**`while`**和带括号的条件，从而形成另一条语句。对应这一语句形成规则的产生式为

<语句> → **while**（条件）<语句>

另一种构建语句的方式是通过选择语句。这些语句具有两种形式，取决于是否有`else`部分，它们可以用以下两个产生式表示

<语句> → **if**（条件）<语句>

<语句> → **if**（条件）<语句>**else**<语句>

还有其他的语句形成方式，比如`for`语句、`repeat`语句和`case`语句。这里将表示它们的产生式留作本节习题，它们从实质上讲与我们已经看到的产生式是类似的。

不过，还有另一种重要的形成规则——程序块，它与我们已经看到的那些多少有些区别。程序块是由花括号`{`和`}`包围0条或更多语句构成的。要描述程序块，就需要一个补充语法分类，这里将其称为<语句列>，它代表一列语句。对应<语句列>的产生式很简单，就是

<语句列> → ϵ

<语句列> →<语句列> <语句>

也就是说，第一个产生式说明语句列可以为空。而第二个产生式则表示如果在一列语句后再加上另一条语句，还是会得到一列语句。

现在就可以定义由语句列围上{和}构成的程序块语句了，也就是

<语句> →（<语句列>）

我们已经给出的这些产生式，加上陈述了语句可以是简单语句（赋值、调用、输入/输出或跳转）跟上分号的依据产生式，就如图11-5所示。

<语句> → **while**（条件）<语句>
<语句> → **if**（语句）<语句>
<语句> → **if**（语句）<语句>**else**<语句>
<语句> → {<语句列>}
<语句> → 简单语句；

<语句列> → ϵ
<语句列> → <语句列><语句>

图11-5　定义了C语言中某些语句形成方式的产生式

11.2.2　习题

(1) 为所有属于C语言标识符的字符串给出定义了语法分类<标识符>的文法。大家会看到，定义一些像<数码>这样的辅助语法分类是很实用的。

(2) C语言中的算术表达式可以接受标识符和数字作为操作数。修改图11-2中的文法，使得操作数也可以是标识符。使用习题(1)中得到的文法来定义标识符。

(3) 数字可以是实数，有着小数点和10的任意乘方，也可以是整数。修改图11-2中表示表达式的文法，或修改习题(2)中写出的文法，允许实数作为操作数。

(4) * C语言算术表达式的操作数还可以是涉及指针（*和&运算符）的表达式，记录结构体的字段（.和->运算符）或数组索引。数组的索引可以是任意表达式。

(a) 为用来定义由一对方括号包围表达式构成的字符串的语法分类<数组引用>写出相应文法。可以利用语法分类<表达式>作为辅助。

(b) 为用来定义作为操作数的字符串的语法分类<名字>写出相应文法。就像1.4节中介绍过的那样，`(*a).b[c][d]`就是个名字。可以利用语法分类<数组引用>作为辅助。

(c) 为允许使用名字作为操作数的算术表达式写出相应文法。可以使用语法分类<名字>作为辅助。在将(a)、(b)、(c)这3个小题得到的产生式结合在一起后，能否得到允许`a[b.c][*d]+e`这种表达式存在的文法？

(5) * 证明图11-4的文法可以生成2.6节中定义的量变平衡括号串。提示：与2.6节中的证明过程类似，要两次用到对括号串长度的归纳。

(6) * 有时候表达式可以有两种或更多种平衡括号。例如，C语言表达式可以同时具有圆括号和方括号，而且两种括号肯定都是平衡的，也就是说，每个`(`都必须匹配一个`)`，而每个`[`都必须匹配一个`]`。为具有这两种类型括号的平衡括号串写出相应文法。也就是说，该文法生成的平衡括号串只能是格式标准的C语言表达式中可能出现的那些。

(7) 为图11-5中的文法添加定义`for`语句、`do-while`语句和`switch`语句的产生式。可以恰当地使用抽象终结符和辅助语法分类。

(8) * 扩展示例11.3中的抽象终结符条件，以体现逻辑运算符的使用。也就是说，定义语法分类<条件>取代抽象终结符条件。可以使用抽象终结符比较表示`x+1<y+z`这样的比较表达式，然后以利用<这样的比较运算符表示的语法分类<比较>和语法分类<表达式>替代抽象终结符比较。语法分类<表达式>可以像11.2节开头那样粗略定义，不过还要加上C语言中的其他运算符，比如一元减号和`%`。

(9) **写出可以定义语法分类<简单语句>的产生式，替换图11-5中的抽象终结符简单语句。可以假设语法分类<表达式>代表了C语言算术表达式。回想一下，简单语句可以是赋值、函数调用或跳转语句，而且严格来说，空串也是简单语句。

11.3 源自文法的语言

文法从本质上讲是涉及字符串集合的归纳定义。2.6节中那些归纳定义的示例与11.2节中很多例子的主要区别在于，对文法而言，一种文法定义若干语法分类的情况很常见。而2.6节中各个例子都定义了单独的概念。虽然存在这种区别，但2.6节中用来构建已定义对象集合的方法也适用于文法。对某文法的各语法分类$<S>$而言，可以按照如下方式定义语言$L(<S>)$。

依据。首先假设对文法中的各语法分类$<S>$而言，语言$L(<S>)$为空。

归纳。假设该文法具有产生式$<S> \to X_1X_2\cdots X_n$，其中对$i=1$、2、…、n，各个X_i要么是语法分类，要么是终结符。并且对$i=1$、2、…、n，按照如下方式为各个X_i选择一个字符串s_i。

(1) 如果X_i是终结符，就可以只使用X_i作为字符串s_i。

(2) 如果X_i是语法分类，就可以选择任何一个已知在$L(X_i)$中的字符串作为s_i。如果若干个X_i是相同的语法分类，就可以从各次出现的$L(X_i)$中分别选不同的字符串作为s_i。

那么所选的这些字符串的串接$s_1s_2\cdots s_n$就是语言$L(<S>)$中的字符串。请注意，如果$n=0$，就把ϵ放到该语言中。

实现这一定义的一种系统化方法是经过该文法各产生式若干轮。在每轮中，我们要以所有可能的方式利用归纳规则更新各语法分类的语言。也就是说，对各个属于语法分类的X_i，要以所有可能的方式从$L(X_i)$中选出字符串。

✦ 示例 11.4

考虑一种由示例11.3介绍的与某几种C语言语句对应的文法中的一些产生式组成的文法。简单起见，我们只会用到对应`while`语句、程序块和简单语句的产生式，以及对应语句列的两个产生式。此外，还要使用简略表示法来合理压缩这些字符串的长度。简略表示用到了终结符**w**（while）、**c**（带括号的条件）和**s**（简单语句）。该文法使用语法分类$<S>$表示语句，并使用语法分类$<L>$表示语句列。该文法的产生式如图11-6所示。

$$
\begin{array}{lll}
(1) & <S> & \to \mathbf{w}\ \mathbf{c}\ <S> \\
(2) & <S> & \to \{\ <L>\ \} \\
(3) & <S> & \to \mathbf{s}\ ; \\
(4) & <L> & \to\ <L>\ <S> \\
(5) & <L> & \to\ \epsilon
\end{array}
$$

图11-6 简化过的表示语句的文法

设L是语法分类$<L>$中字符串的语言，并设S是语法分类$<S>$中字符串的语言。一开始，根据依据规则，L和S都为空。在第一轮中，只有产生式(3)和产生式(5)是有用的，因为其他产生式的

右部都含有语法分类，而此时对应这些语法分类的语言中还没有任何字符串。产生式(3)表明了**s;**是语言S中的字符串，而产生式(5)则告诉我们ϵ 在语言L中。

第二轮开始时，有$L = \{\epsilon\}$和$S = \{$**s;**$\}$。产生式(1)现在让我们可以把**wcs;**添加到S中，因为**s;**已经在S中了。也就是说，在产生式(1)的右部中，终结符**w**和**c**只能代表它们本身，语法分类<*S*>则可以被替换为语言S中的任意字符串。因为目前**s;**是S中唯一的字符串，所以我们只能作出这一个选择，而这一选择的结果就是**wcs;**。

产生式(2)添加了字符串**{}**，因为终结符**{**和**}**只能代表它们自身，不过语法分类<*L*>可以代表语言L中的任意字符串，而此刻L中只含有ϵ 。

因为产生式(3)的右部只由终结符构成，所以它决不会产生除**s;**之外的字符串，因此从现在开始就可以忘掉该产生式了。同样，产生式(5)也不会产生除ϵ 之外的字符串，所以在本轮及以后的轮次中也可以忽略它。

最后，当我们用ϵ 代替<*L*>并用**s;**代替<*S*>之后，产生式(4)就为L产生了字符串**s;**。在第二轮结束的时候，语言 $S=\{$**s; wcs; ,{}**$\}$，而语言$L = \{\epsilon$; **s;**$\}$。

在下一轮中，可以使用产生式(1)、(2)和(4)产生新的字符串。产生式(1)中替换<S>的选择有3种，分别是**s;**、**wcs;**和**{}**。用**s;**替换后得到的字符串是语言S中已经具有的，不过替换了另两个字符串后得到的是新字符串**wcwcs;**和**wc{}**。

对产生式(2)而言，可以用ϵ 或**s;**替换<*L*>，为语言S得出旧字符串**{}**和新字符串**{s;}**。在产生式(4)中，可以用ϵ 或**s;**替换<*L*>，并用**s;**、**wcs;**或**{}**替换<*S*>，这给语言L带来了一个旧字符串**s;**，以及5个新字符串**wcs;**、**{}**、**s;s;**、**s;wcs;**和**s;{}**。[①]

这样语言就成了 $\boldsymbol{S}=\{\textbf{s;,wcs;,\{\},wcwcs;,wc\{\},\{s;\}}\}$ 和 $\boldsymbol{L}=\{\epsilon,\textbf{s;,wcs;,\{\},s;s;,s;wcs;,s;\{\}}\}$ 。我们可以按照自己的意愿继续这一过程。图11-7总结了前3轮的情况。

	S	L
Round 1:	`s;`	ϵ
Round 2:	`wcs;` `{}`	`s;`

Round 3:	`wcwcs;`
	`wc{}`
	`{s;}`
		`wcs;`
		`{}`
		`s;s;`
		`s;wcs;`
		`s;{}`

图11-7　前3轮每轮产生的新字符串

正如示例11.4中那样，由文法定义的语言可能是无限的。如果语言是无限的，我们就不能一一列出所有字符串。能做到的最佳做法就是按轮次枚举这些字符串，就像示例11.4中一开始所做的那样。该语言中的任何字符串都将在某轮出现，不过在任何一轮都不可能产生出所有的字符串。可以被放进语法分类<*S*>的语言中的那些字符串组成的集合，就形成了（无限）语言L(<*S*>)。

① 我们非常关心用字符串替换语法分类的方式。假设经过每一轮替换，语言L和S都和它们在上一轮结束时的定义保持不变。每个产生式的右部都要进行替换。这些右部可以为左部的语法分类产生新的字符串，不过在同一轮替换中，我们不会在一个产生式的右部使用由另一个产生式新构建的字符串。但这也没关系，所有要生成的字符串最终都会在某一轮中生成，不管我们是立即在右部中循环使用新字符串，还是等待在下一轮中再使用新字符串。

习题

(1) 在示例11.4中，第四轮添加的新字符串都有哪些？

(2) * 在示例11.4的第i轮替换中，为各语法分类产生的新字符串中最短的字符串各是什么？为下列语法分类产生的最长新字符串又分别是什么？

(a) <S>

(b) <L>？

(3) 分别使用下列图中的文法按轮次生成平衡括号串。

(a) 图11-3

(b) 图11-4

这两种文法在相同轮次中是否会生成相同的平衡括号串？

(4) 假设每个以某语法分类<S>为其左部的产生式在其右部中也含有<S>，那么为什么L(<S>)为空？

(5) * 在如本节所描述的方式按轮次生成字符串时，为语法分类<S>生成的新字符串只可能是通过替换某有关<S>的产生式中右部的语法分类得到，*满足至少有一个被替换的字符串是在上一轮中新发现的*。解释一下为什么楷体标记的条件是正确的。

(6) ** 假设我们想分辨某个特定的字符串s是否在某语法分类<S>的语言中。

(a) 解释一下，为什么如果在某轮中所有为任意语法分类生成的新字符串都比s长，而且s尚未为L(<S>)生成，s就不可能被放入L(<S>)中。*提示*：利用习题(5)。

(b) 解释一下，为什么在有限轮次的替换后，一定没法继续生成不长于s的新字符串。

(c) 使用(a)和(b)的结论设计一种算法，接受某文法、该文法的一种语法分类<S>，以及某个终结符串s，并分辨出s是否在L(<S>)中。

11.4　分析树

正如我们已经看到的，通过反复应用产生式，可以为某语法分类<S>得出字符串s属于语言L(<S>)的结论。从由右部中不含语法分类的依据产生式得到的字符串开始。然后，对已经从各语法分类得到的字符串“应用”产生式。每次应用都要用字符串替换产生式右部中出现的各语法分类，并构造出属于产生式左部中语法分类的字符串。最终，我们将通过应用左部为<S>的产生式来构造字符串s。

把s在L(<S>)中的“证明”画成一棵称作*分析树*（parse tree）的树往往是很实用的。分析树的节点都是带标号的，要么是终结符，要么是语法分类，要么是符号ϵ。叶子节点只会被标记为终结符或符号ϵ，而内部节点只可能用语法分类作为标号。

每个内部节点v都表示产生式的应用。也就是说，一定存在某个产生式同时满足下列条件：

(1) 标号v的语法分类是该产生式的左部；

(2) v的子节点的标号从左往右构成了该产生式的右部。

✦　示例 11.5

图11-8展示了一棵基于图11-2所示文法的分析树。不过，在这里我们把语法分类<*表达式*>、<*数字*>和<*数码*>分别简称为<E>、<N>和<D>。该分析树表示的字符串是**`3*(2+14)`**。

例如，这棵分析树的根节点及其子节点就表示产生式

$$<E> \to <E> * <E>$$

就是图11-2中的产生式(6)。根节点的最右子节点及其子节点形成了产生式<E> → (<E>)，或者说是图11-2中的产生式(5)。

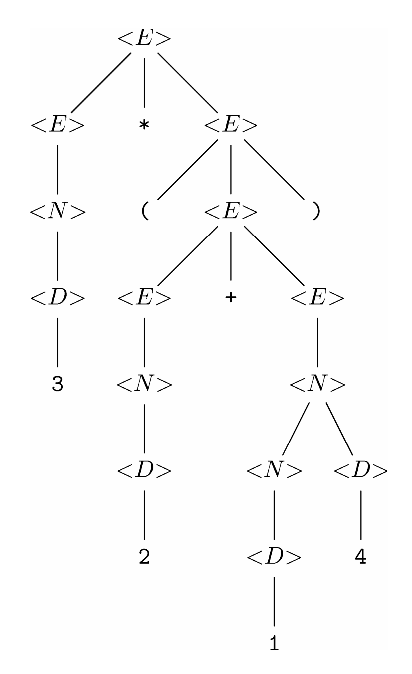


图11-8　使用图11-2所示文法的字符串**3*(2+14)**对应的分析树

11.4.1　分析树的构建

每棵分析树都表示某一终结符串s，我们可将该串称为这棵树的*产出*（yield）。串s由相应分析树所有叶子节点的标号按照从左到右的次序排列而成。此外，通过对分析树进行前序遍历并只依次列出那些属于终结符的标号，我们也可以得到这一产出。例如，图11-8所示分析树的产出就是**3*(2+14)**。

如果树只有一个节点，那么该节点的标号就只能是某个终结符或者ϵ，因为它是个叶子节点。如果该树不止有一个节点，那么根节点的标号就是语法分类，因为在一棵有两个或更多个节点的树中，根节点总是个内部节点。而且该语法分类的字符串中总是会包含该树的产出。与某给定文法对应的分析树的归纳定义如下所述。

依据。对文法中的每个终结符**x**来说，存在一棵只含一个标号为**x**的节点的树。当然，该树的产出就是**x**。

归纳。假设我们有产生式$<S>\to X_1X_2\cdots X_n$，其中各个X_i要么是终结符，要么是语法分类。如果$n=0$，也就是说，该产生式实为$<S>\to\epsilon$，那么就有一棵像图11-9这样的树。其产出为ϵ，而且根节点为$<S>$，因为有该产生式，所以空串ϵ显然是在$L(<S>)$中的。

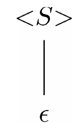


图11-9　由产生式$<S>\to\epsilon$得到的分析树

现在假设$<S>\to X_1X_2\cdots X_n$而且$n\geqslant 1$。我们可以按照如下方式，对每个$i=1$、2、…、n而言，为各个X_i选择树T_i。

(1) 如果 X_i 是终结符，就必须选择标号为 X_i 的单节点树。如果有两个或多个X是同一终结符，就必须为该终结符的每次出现选择具有相同标号的不同单节点树。

(2) 如果 X_i 是语法分类，我们可以选择任何已经构建好的以 X_i 作为根节点标号的分析树，然后构建一棵像图11-10这样的树。也就是说，我们创建的根节点标号是该产生式左部的语法分类<*S*>，而这棵树根节点的子节点从左到右依次是为X_1、X_2、…、X_n选择的树的根节点。如果有两个或多个*X*是相同的语法分类，我们也许要为各语法分类选择相同的树，但是必须在该树每次被选中时为其生成不同的副本。我们还可以为同一语法分类的不同出现选择不同的树。

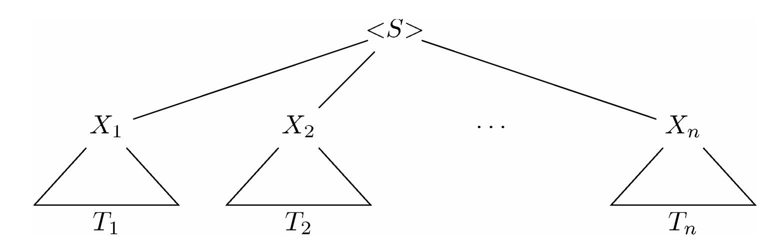


图11-10　利用产生式和其他分析树构建分析树

✦ 示例 11.6

我们来研究一下图11-8中分析树的构造，看看它的结构是如何模仿证明字符串**`3*(2+14)`**在*L*(<*E*>)中的过程的。首先，可以为该树中的各个终结符构造一棵单节点树。然后图11-2中第(1)行的产生式组说明了10个数码都是属于*L*(<*D*>)的长度为1的字符串。我们用到其中的4个产生式创建图11-11所示的4棵树。例如，我们利用产生式<*D*>→1按照如下方式创建了图11-11a中的分析树，为右部中的符号**`1`**创建一棵只有一个标号为`1`的节点的树，然后，创建一个标号为<*D*>的节点作为根节点，并以我们为**`1`**选择的树的根节点（也是唯一的节点）作为其子节点。

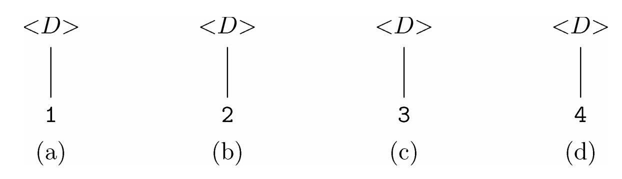


图11-11　使用产生式<*D*>→1以及相似的产生式构建的分析树

下一步是要利用图11-2中的产生式(2)，或者说是<*N*>→<*D*>，来揭示数码就是数字这一事实。例如，可以选择图11-11a所示的树替换产生式(2)右部中的<*D*>，得出图11-12a所示的树。图11-12中的另两棵树也是用相似的方式产生的。

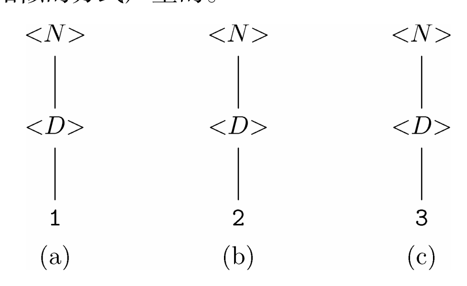


图11-12　使用产生式<*N*>→<*D*>构建的分析树

现在可以利用产生式(3)，也就是<N> → <N> <D>了。我们将会为右部中的<N>选择图11-12a所示的树，并为<D>选择图11-11d所示的树。还要为左部创建一个标号为<N>的新节点，并为该节点指定两个子节点，也就是选中的两棵树的根节点。得到的树如图11-13所示。该树的产出是数字`14`。

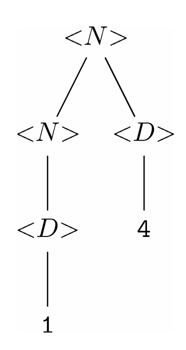


图11-13　用产生式<N> → <N> <D>构建的分析树

下一个任务就是为和`2+14`创建分析树。首先，我们要用到产生式(4)，即<E> → <N>，以建立图11-14所示的分析树。这些树表明了`3`、`2`和`14`都是表达式。这些树中的第一棵源自图11-12c中为右部中的<N>选择的树，第二棵是通过图11-12b中为<N>选择的树得到的，而第三棵则是选择图11-13中的树得到的。

然后可以使用产生式(6)，也就是<E> → <E> + <E>。对右部中的第一个<E>，我们使用了图11-14b中的树，而对右部中的第二个<E>，则是使用了图11-14c所示的树。为右部中的+使用的是一棵标号为+的单节点树。得到的树如图11-15所示，其产出为`2+14`。

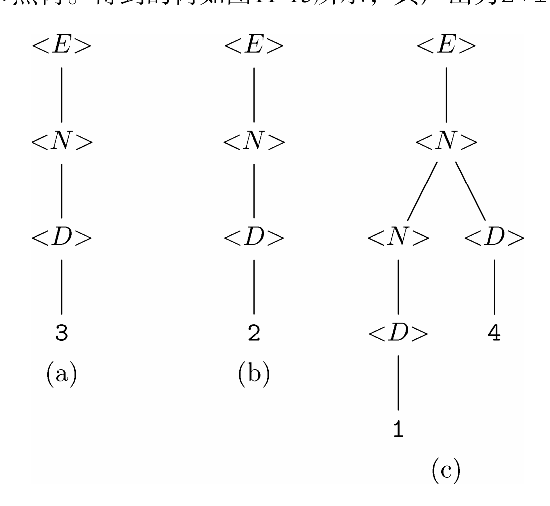


图11-14　使用产生式<E> → <N>构建的分析树

接下来要用到产生式(5)，或者说是<E> →(<E>)，构建图11-16所示的分析树。我们只要为右部中的<E>选择图11-15中的树，并为终结符括号选择单节点树即可。

最后，利用产生式(8)，也就是<E> → <E> * <E>，构建了我们最初在图11-8中展示的分析树。我们为右部中的第一个<E>选择了图11-14a所示的树，并为第二个<E>选择了图11-16中的树。

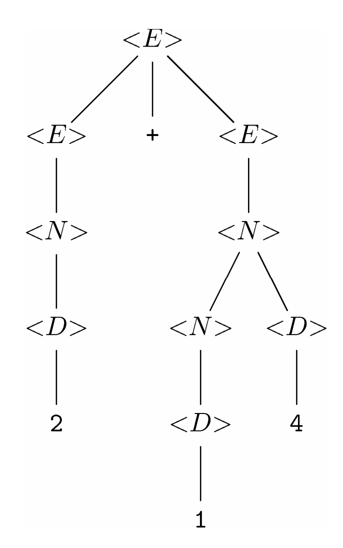


图11-15　使用产生式<*E*> → <*E*> + <*E*>构建的分析树

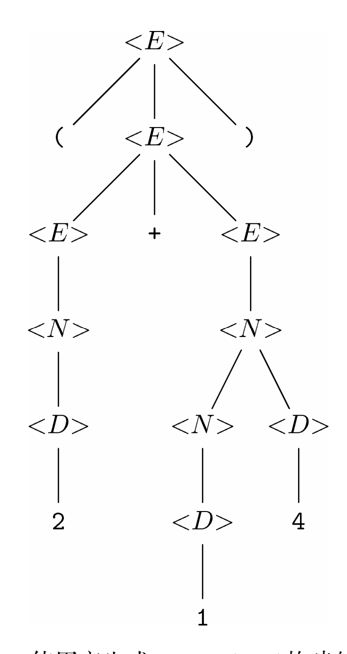


图11-16　使用产生式<*E*> → (<*E*>)构建的分析树

11.4.2　分析树为何“行得通”

分析树的构建与字符串属于某语法分类的归纳定义非常相似。我们可以通过两次简单的归纳来证明，对任意语法分类<*S*>来说，以<*S*>为根节点的分析树的产出刚好是*L*(<*S*>)中的字符串。也就是如下两点。

(1) 如果*T*是根节点标号为<*S*>而且产出为*s*的分析树，那么字符串*s*在语言*L*(<*S*>)中。

(2) 如果字符串*s*在语言*L*(<*S*>)中，那么存在产出为*s*且根节点标号为<*S*>的分析树。

这一等价关系应该是相当直观的。粗略地讲，分析树是由更小的分析树，按照由较短的字

符串构成长字符串的方式，对产生式右部中的语法分类进行替换构成的。我们首先利用对树T高度的完全归纳证明第(1)部分。

依据。假设分析树的高度是1。那么这棵树就像图11-17所示的这样，或者，在$n=0$的特例中，就像图11-9所示的树那样。构建这种树的唯一方法是，若存在产生式<*S*>→$\mathbf{x}_1\ \mathbf{x}_2\cdots\mathbf{x}_n$，其中各**x**都是终结符(如果$n=0$，该产生式就是<*S*>→$\epsilon$)。因此$\mathbf{x}_1\ \mathbf{x}_2\cdots\mathbf{x}_n$是$L$(<*S*>)中的字符串。

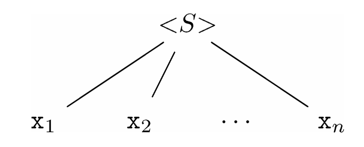


图11-17　高度为1的分析树

归纳。假设命题(1)对所有高度不超过k的树都成立。现在考虑像图11-10那样高度为k+1的树。那么，对$i=1$、2、…、n，各子树T_i的高度至多为k。如果这些子树中有任何一棵的高度达到或超过k+1，那么整棵树的高度就至少是k+2。因此，归纳假设适用于各棵树T_i。

根据归纳假设，如果子树T_i的根节点X_i是语法分类，那么T_i产出s_i就在语言$L(X_i)$中。如果X_i是终结符，就定义字符串s_i是X_i，那么整棵树的产出就是$s_1s_2\cdots s_n$。

根据分析树的定义，可知<*S*> →$X_1\ X_2\cdots X_n$是产生式。假设只要X_i是语法分类就用s_i替换X_i。根据定义，如果X_i是终结符，X_i就是s_i。这样一来，替换后的右部就成了$s_1s_2\cdots s_n$，与该树的产出是相同的。根据<*S*>的语言的归纳规则，我们知道$s_1s_2\cdots s_n$是在L(<*S*>)中的。

现在必须证明命题(2)，语法分类L<*S*>中的每个字符串s都具有以<*S*>为根节点且以s为产出的分析树。首先要注意到，对每个终结符x，存在根节点和产出都是x的分析树。现在我们要对得出s在L(<*S*>)中时的归纳步骤（如11.3节所述，下面的证明中加引号的“归纳步骤”就是表示该归纳步骤）的应用次数进行完全归纳。

依据。假设证明s在L(<*S*>)中需要应用“归纳步骤”一次。则一定存在产生式<*S*>→$\mathbf{x}_1\ \mathbf{x}_2\cdots\mathbf{x}_n$，其中所有的**x**都是终结符，而且<*S*> = $\mathbf{x}_1\ \mathbf{x}_2\cdots\mathbf{x}_n$。我们知道对$i=1$、2、…、$n$，都有标号为$\mathbf{x}_i$的单节点分析树。因此，存在产出为$s$且根节点标号为<*S*>的分析树，该树的样子类似图11-17所示。在$n=0$的特例中，我们知道s = ϵ，此时就要使用图11-9所示的树。

归纳。假设应用“归纳步骤”不超过k次所发现的任意语法分类<*T*>的语言中，任何字符串t都具有以t为产出而且以<*T*>为根节点的分析树。考虑通过k+1次应用“归纳步骤”找到的在语法分类<*S*>的语言中的字符串s。那么，存在产生式$<S>\rightarrow X_1X_2\cdots X_n$，且$s=s_1s_2\cdots s_n$，其中每个子串$s_i$都会是如下两种可能之一。

(1) 为X_i（如果X_i是终结符）。

(2) 某个至多应用k次“归纳步骤”就可知在$L(X_i)$中的字符串（如果X_i是语法分类）。

因此，对每个i，都可以找到一棵具有产出s_i而且根节点标号为X_i的树T_i。如果X_i是语法分类，那么就利用归纳假设声明T_i存在，而如果X_i是终结符，则不需要归纳假设就可以声明存在标号为X_i的单节点树。因此，如图11-10中那样构建的树具有产出s而且根节点标号为<*S*>，这样就证明了该归纳步骤。

11.4.3　习题

(1) 根据图11-2所示的文法，为以下字符串给出相应的分析树。在各情况中根节点位置的语法分类都

应该是<*E*>。

(a) 35+21

(b) 123-(4*5)

(c) 1*2*(3-4)

(2) 使用图11-6中的语句文法，给出以下字符串的分析树。其中每种情况下根节点的语法分类都应该是<*S*>。

(a) wcwcs;

(b) {s;}

(c) {s;wcs;}

(3) 利用图11-3中的平衡括号串文法，给出以下括号串的分析树。

(a) (()())

(b) ((()))

(c) ((()))())

(4) 利用图11-4中的文法为习题(3)中的各括号串给出分析树。

语法树和表达式树

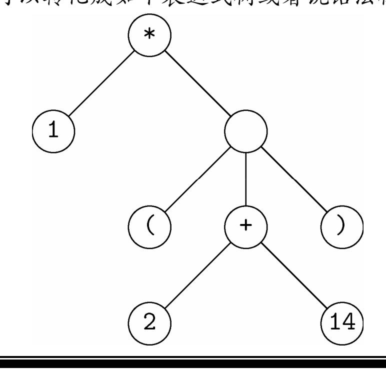
通常，像分析树这样的树是用来表示表达式的。例如，我们在第5章中通篇都以表达式树为例。语法树是“表达式树”的另一个名字。当我们拥有如图11-2所示的对应表达式的文法时，就可以通过以下3项变形把分析树转换成表达式树。

(1) 原子操作数被压缩为以该操作数为标号的单个节点。

(2) 运算符从叶子节点移动到它们的父节点。也就是说，像+这样的运算符符号成为了原本在它上方，以语法分类“表达式”为标号的节点的标号。

(3) 仍然以“表达式”为标号的内部节点要删除其标号。

例如，图11-8中的分析树就可以转化成如下表达式树或者说语法树。

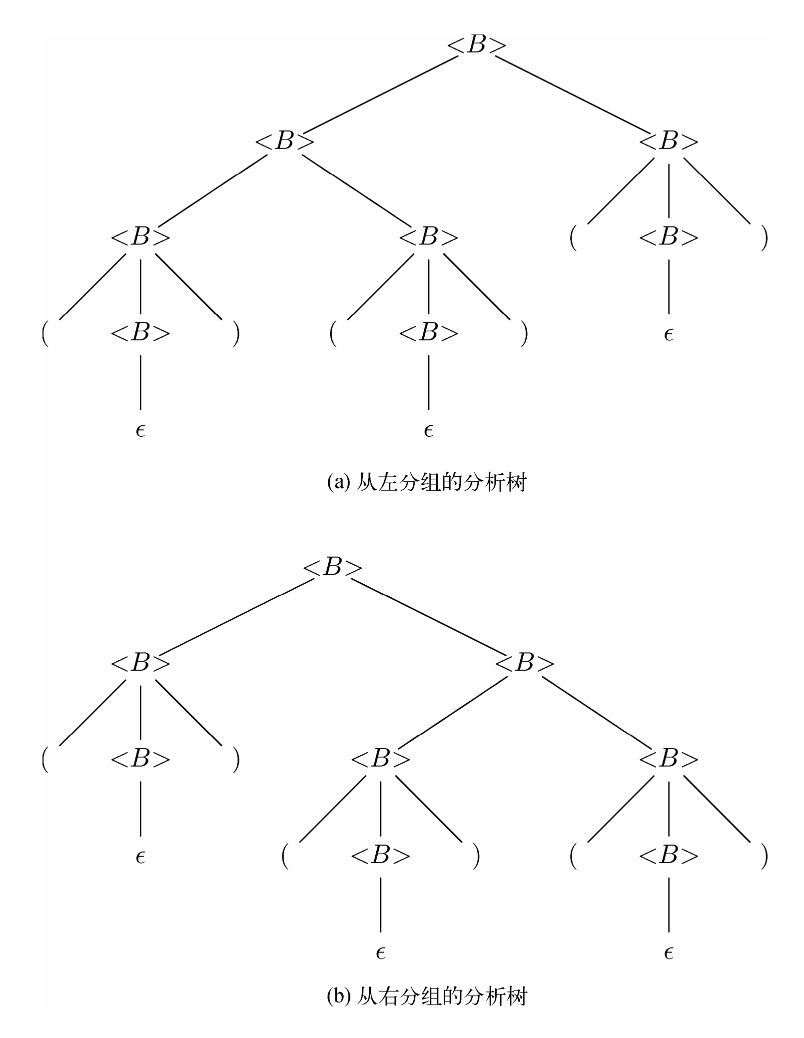


11.5　二义性和文法设计

我们来考虑如图11-4所示的表示平衡括号串的文法，这里用语法分类<*B*>作为图11-4中语法分类<平衡>的缩写。

$$<B> \to (<B>) \mid <B><B> \mid \epsilon \qquad (11.1)$$

假设想要一棵表示括号串()()()的分析树。图11-18展示了两棵这样的分析树，第一棵是把前两对括号先分在一组，而另一棵则先把后两对括号分成一组。



(a) 从左分组的分析树

(b) 从右分组的分析树

图11-18　有着同样的产出和根节点的两棵分析树

出现这样两棵分析树应该不会让人感到惊讶。一旦确定 ( ) 和 ( ) ( ) 都是平衡括号串，使用产生式 $< B > \to < B > < B >$ 就可以用 ( ) 替换右部中的第一个<*B*>并用 ( ) ( ) 替换第二个<*B*>，反之亦然。两种情况下，都能得出括号串 ( ) ( ) ( ) 在语法分类<*B*>中。

如果文法中有两棵或多棵分析树具有相同产出，且其根节点标号是相同的语法分类，就说该文法是*二义的*（ambiguous）。请注意，不一定要每个字符串都是若干分析树的产出，只要有一个这样的字符串就足够让文法具有二义性了。例如，括号串 ( ) ( ) ( ) 就足以说明文法(11.1)是二义的。不具二义性的文法叫作*无二义*（unambiguous）文法。在无二义文法中，对每个字符串*s*和语法分类<*S*>而言，至多存在一棵产出为*s*且根节点标号为<*B*>的分析树。

图11-3所示的文法就是个无二义文法的例子，这里还是用<*S*>来代替<*平衡的*>。

$$< B > \to (< B >) < B > | \epsilon \qquad (11.2)$$

证明文法无二义是相当困难的。在图11-19中是对应括号串 ( ) ( ) ( ) 的唯一一棵分析树，当然，这一字符串有着唯一分析树的事实并不能证明文法(11.2)就是无二义的。我们证明无二义性的唯一方法就是证明语言中的每个字符串都具有唯一的分析树。

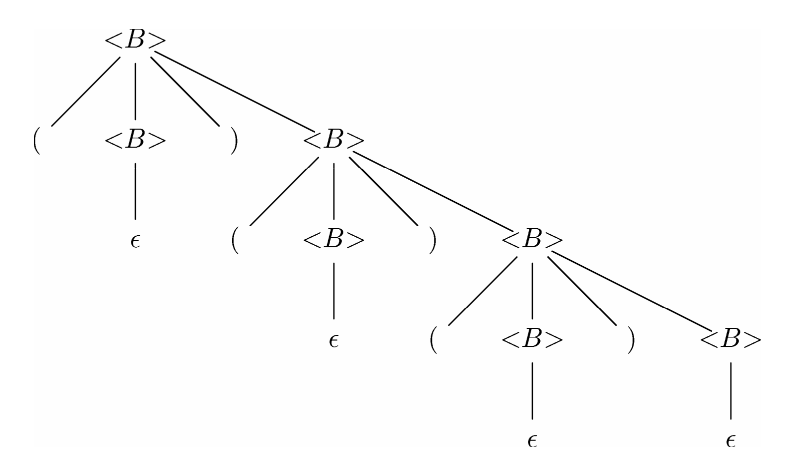


图11-19　使用文法(11.2)表示字符串 ( ) ( ) ( ) 的唯一分析树

11.5.1　表达式中的二义性

尽管图11-4中的文法是二义的，但它的二义性并没有太大坏处，因为我们从左起还是从右起分组括号影响不大。在考虑表示表达式的文法（比如11.2节中图11-2所示的文法）时，会发生一些更为严重的问题。具体地讲，尽管一些分析树可以给出正确的表达式值，但另一些分析树表示的是错误的值。

无二义性为何很重要

为程序构建分析树的分析器是编译器的关键部分。如果描述编程语言的文法是二义的，而且如果其二义性未被消除，那么就至少有某些程序具有多棵分析树。而同一程序不同的分析树就为该程序赋予了不同的含义，其中这种情况下“含义”是指由原始程序翻译成的机器语言程序执行的操作。因此，如果与程序对应的文法是二义的，编译器就不能正确地决定该为某些程序使用哪棵分析树，所以就不能决定机器语言程序应该做些什么。出于这种原因，编译器必须使用无二义性的规范。

✦ 示例 11.7

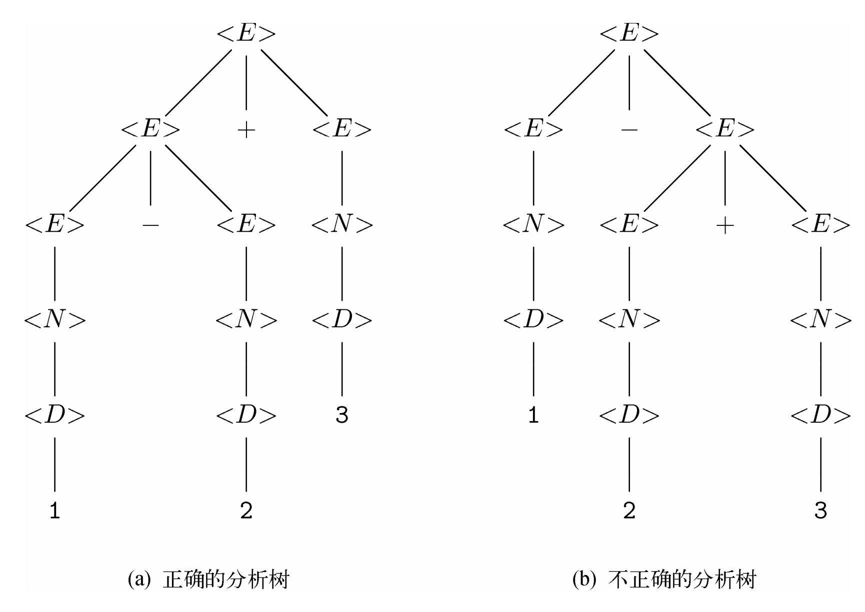
这里用简略表示法来表示示例11.5中给出的表达式文法，并考虑表达式`1-2+3`。它具有两棵分析树，取决于是从左还从右组合运算符。这两棵分析树如图11-20a和图11-20b所示。

图11-20a中的树是从左起结合的，因此操作数是从左起分组的。这种分组是正确的，因为我们一般会从左起分组优先级相同的运算符，**`1-2+3`**习惯被解释为**`(1-2)+3`**，其值为2。如果我们为构建起图11-20a所示树的子树表示的表达式求值，就要首先在根节点的最左子节点处计算$1-2=-1$，然后在根节点计算$-1+3=2$。

另一方面，对从右侧起关联的图11-20b，会把该表达式分组为**`1-(2+3)`**，其值为–4。不过，对该表达式的这种解释是不合规定的。值–4是在构建图11-20b的树时得到的，因为我们先在根节点的最右子节点处计算了$2+3=5$，然后在根节点处计算了$1-5=-4$。

从错误的方向结合优先级相等的运算符可能导致问题。而优先级不同的运算符也可能带来

问题，正如我们在接下来的示例中要看到的，有可能在结合更高优先级的运算符之前先结合了低优先级的运算符。

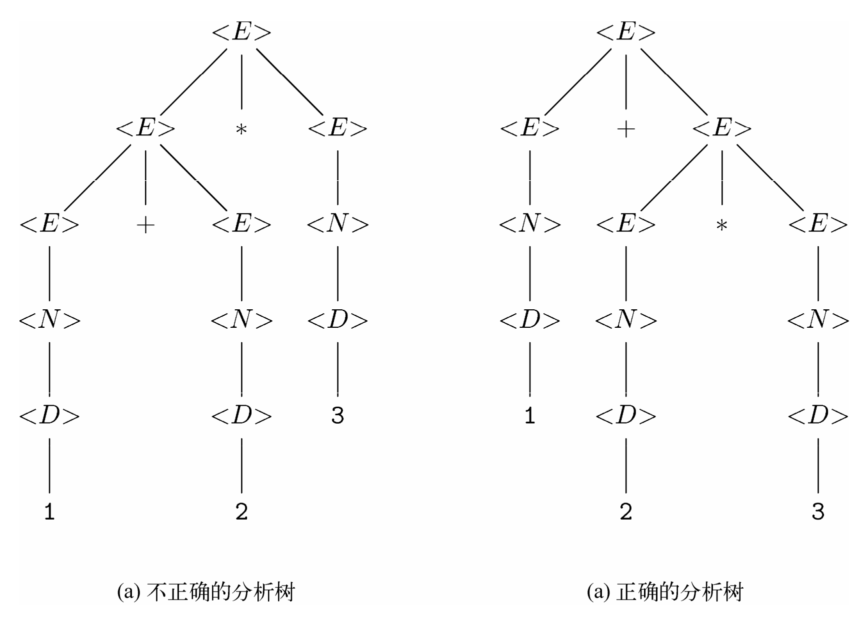


(a) 正确的分析树　　(b) 不正确的分析树

图11-20　对应表达式1-2+3的两棵分析树

✦ 示例 11.8

考虑表达式1+2*3。在图11-21a中，我们看到表达式是从左起分组的，这是不对的，而图11-21b所示的则是正确的从右边起的分组，这样乘法的操作数才在加法之前分组。前一种分组会得出不正确的值9，而后面的分组则会产生合乎规则的值7。



(a) 不正确的分析树　　(a) 正确的分析树

图11-21　表示表达式1+2*3的两棵分析树

11.5.2 表示表达式的无二义文法

就像表示平衡括号串的文法(11.2)可以被视作文法(11.1)的无二义版本那样，也可以为示例11.5中的表达式文法构建一个无二义版本。“窍门”就是定义有着如下直觉含义的3个语法分类。

(1) <因式>生成了不能被“提取出”的表达式，也就是说，因式要么是单个操作数，要么是加了括号的表达式。

(2) <项>生成了因式的积或商。单个因式是项，因此一列由*或/运算符分隔的因式也是项。**12**和**12/3**<因式>***45**都是项。

(3) <表达式>生成了一项或多项的和或差。单个项就是个表达式，因此一列由+或–运算符分隔的项也是表达式。**12**、**12/3*45**和**12+3*45-6**都是表达式。

图11-22就是表示表达式、项和因式间关系的文法。我们用简写<*E*>、<*T*>和<*F*>分别代表<因式>、<项>和<表达式>。

$$
\begin{array}{ll}
(1) & <E> \to <E> + <T> \mid <E> - <T> \mid <T> \\
(2) & <T> \to <T> * <F> \mid <T>/<F> \mid <F> \\
(3) & <F> \to (<E>) \mid <N> \\
(4) & <N> \to <N><D> \mid <D> \\
(5) & <D> \to \mathbf{0} \mid \mathbf{1} \mid \cdots \mid \mathbf{9}
\end{array}
$$

图11-22 表示算术表达式的无二义文法

例如，第(1)行的3个产生式定义了表达式要么是较小的表达式后面跟上+或–以及另一项，要么是单独的项。如果将这些概念融为一体，那么该产生式是说，每个表达式都是项后面跟上0个或更多由一个+或–以及一项构成的配对。同样，第(2)行表示项是由较小的项后面跟上*或/以及因式构成的。也就是说，项是由因式后面跟上0个或更多由一个*或/加上一个因式组成的配对。第(3)行说的是因式或者是数字，或者是由括号包围的表达式。而第(4)行和第(5)行则像之前所做的那样定义了数字和数码。

之所以在第(1)行和第(2)行中使用了诸如

$$<E> \to <E> + <T>$$

这样的产生式，而没有使用看似与之等价的$<E> \to <T> + <E>$，就是为了强制这些项从左起分组。因此，我们看到像**1-2+3**这样的表达式会被正确地分组为**(1-2)+3**。同样，像**1/2*3**这样的项也能被正确地分组为**(1/2)*3**。图11-23展示了用图11-22中的文法表示表达式**1-2+3**的唯一分析树。请注意，**1-2**必须首先被组合为表达式。如果像图11-20b中那样先组成**2+3**，是没办法用图11-22所示的文法将**1-**附加到该表达式上的。

表达式、项和因式之间的区别使得处于不同优先级的运算符能被正确分组。例如，表达式**1+2*3**对应的分析树只有图11-24所示的那棵，它像图11-21b所示的树那样先组合了子表达式**2*3**，而不是像图11-21a所示的错误的树那样首先组合**1+2**。

就像之前提到的平衡括号串问题那样，我们没有证明图11-22所示的文法是无二义的。习题中包含了更多例子，应该有助于说服读者相信该文法不仅是无二义的，而且为各个表达式给出了正确的组合方式。我们还表述了该文法的思路如何扩展到更全面的表达式家族。

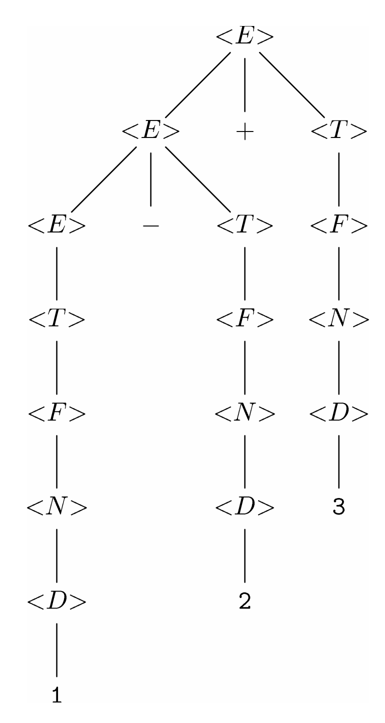


图11-23　用图11-22中的无二义文法表示表达式`1-2+3`的分析树

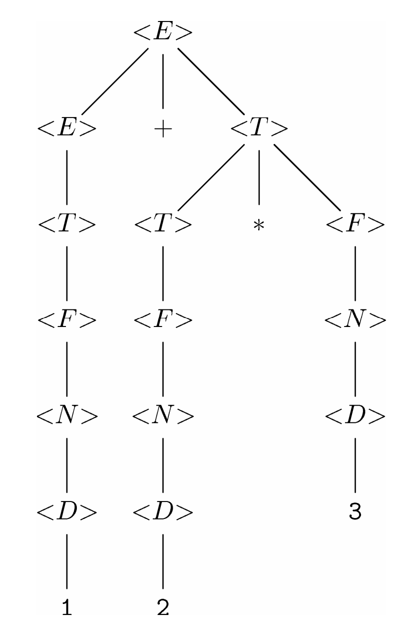


图11-24　用图11-22中的无二义文法表示表达式`1+2*3`的分析树

11.5.3　习题

(1) 用图11-22所示的文法，为下列各表达式给出唯一的分析树。

(a) `(1+2)/3`
(b) `1*2-3`
(c) `(1+2)*(3+4)`

(2) * 图11-22所示文法的表达式有两级优先级，+和–在第一级，而*和/在更高的第二级。一般而言，我们可以利用k+1个语法分类处理具有k个优先级的表达式。修改图11-22中的文法，使其包含乘方运算符^，它的优先级比*和/更高。作为提示，大家要定义是操作数或带括号表达式的*要素*，并重新把*因式*定义为一个或多个要素由乘方运算符连接而成。请注意，乘方的组合是从右起而不是从左起的，也就是说，**2^3^4**表示的是**2^(3^4)**，而不是**(2^3)^4**。我们该如何确保在要素中是从右起进行组合的？

(3) * 扩展该无二义表达式文法，允许**=**、**<=**等比较运算符，这些比较运算符都具有同样的优先级而且是左关联的，它们的优先级在+和–之下。

(4) 扩展图11-22中的表达式文法，使其包含一元减号。请注意，这一运算符的优先级要比其他运算符的优先级更高，例如，**-2*-3**就被组合为**(-2)*(-3)**。

(5) 扩展习题(3)中得出的文法，已包含逻辑运算符**&&**、||和**!**。其中**&&**和*有着相同优先级，而||的优先级与+相同，而**!**的优先级则比一元的–更高。**&&**和||这两个二元运算符都是从左起组合的。

(6) * 不是每个表达式按照11.2节图11-2所示的二义性文法都有一棵以上的分析树。给出一些根据该文法只有唯一分析树的表达式。大家能否给出一条规则，说明什么时候表达式会具有唯一的分析树？

(7) 以下文法定义了只有0和1组成的字符串的集合（不含ϵ）。

<字符串> → <字符串> <字符串> | 0 | 1

在该文法中，字符串010有几棵分析树？

(8) 给出与习题(7)中文法具有相同语言的无二义文法。

(9) * 文法(11.1)对空串来说有多少分析树？给出对应空串的3种不同的分析树。

11.6　分析树的构造

文法和正则表达式一样，都可以描述语言，但都不能直接给出算法来确定某字符串是否在所定义的语言中。对正则表达式来说，我们在第10章中已经了解到如何先把正则表达式转换成非确定自动机，接着转换成确定自动机，而这一确定自动机就可以直接实现为程序。

对文法来说，也存在多少有些相似的处理过程。一般来说，根本不可能把文法转换成确定自动机，在11.7节中我们会讨论一些可以进行这种转换的例子。不过，通常可以把文法转换成类似自动机的程序，从头至尾读取输入，并呈现该输入字符串是否在该文法的语言中的决策。这类技术中最重要的就是“LR分析”（LR代表从左至右扫描输入），但它不在本书要讨论的范围之内。

11.6.1　递归下降分析

这里要介绍的是一种更加简单但不那么强大的分析技术——“递归下降分析”，在这种分析中，文法会被一系列相互递归的函数替代，每个递归函数都对应文法中的一个语法分类。对应语法分类<S>的函数S的目标是读入构成语言L(<S>)中字符串的字符序列，并返回指向该字符串分析树根节点的指针。

产生式的右部可以看作找到左部的语法分类中的字符串所必须满足的一系列目标——终结符和语法分类。例如，考虑表示平衡括号串的无二义文法，我们在图11-25中将其重现。

(1)	$<B> \;\rightarrow\; \epsilon$
(2)	$<B> \;\rightarrow\; (\ <B>\ )\ <B>$

图11-25　表示平衡括号串的文法

产生式(2)表述了弄清平衡括号串是否依次满足以下4个条件的一种方法。

(1) 找到字符(；

(2) 然后找到平衡括号串；

(3) 然后找到字符)；

(4) 最后找到另一个平衡括号串。

一般而言，如果发现某终结符是下一个输入符号，终结符的目标就得到满足了，但如果下一个输入符号是其他内容，这一目标就不会被满足了。要弄清右部中的语法分类是否得到满足，可以调用对应该语法分类的函数。

根据文法构建分析树的安排如图11-26所示。假设要确定某终结符序列 $X_1X_2\cdots X_n$ 是否为语法分类<*S*>中的字符串，而且如果是还要给出它的分析树。然后我们在输入文件中放入 $X_1X_2\cdots X_n$ `ENDM`，其中`ENDM`是一个不属于终结符的特殊符号。[①]`ENDM`叫作*端记号*（endmarker），它的作用是表示待检查的整个字符串已经被读入了。例如，在C语言程序中，通常会使用EOF或EOS字符作为端记号。

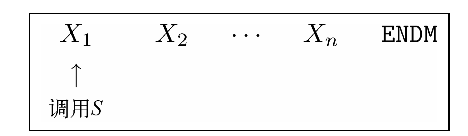


图11-26　初始化在输入中发现<*S*>的程序

输入游标（input cursor）标记了要被处理的终结符，也就是*当前的终结符*。如果输入是字符串，那么游标可以是指向字符的指针。分析程序首先要调用对用语法分类<*S*>的函数*S*，而且输入游标是在输入的开头位置。

每当处理产生式右部，并在产生式中遇到终结符`a`的时候，就要在输入游标指示的位置查找相匹配的终结符`a`。如果找到`a`，就把输入游标移至输入中的下一个终结符。如果当前的终结符是`a`之外的内容，就是匹配失败，就不能为该输入字符串给出分析树。

另一方面，如果处理产生式右部并遇到了语法分类<*T*>，就要调用与<*T*>对应的函数*T*。如果*T*"失败"，那么整个分析也失败，而该输入就被视为不在待分析的语言中。如果*T*成功，它就会"消灭"某一输入，把输入游标向前移动对应该输入的0个或更多位置。从*T*被调用时的位置直到*T*离开游标之前的位置都要被销毁。*T*还会返回一棵树，就是与该被销毁输入对应的分析树。

当我们处理完产生式右部中的各个符号后，就要为该产生式表示的那部分输入生成分析树。要完成这一工作，就需要创建一个新的根节点，并以该产生式的左部作为其标号。该根节点的子节点是成功调用与右部中语法分类对应的函数所返回的树的根节点，而且要为右部中的每个终结符创建相应的叶子节点。

---

① 在实际用于编程语言的编译器中，整个输入可能不是一次性放入一个文件中的，而是由每次检查源程序中一个字符的"词法分析器"这种预处理器一次找到一个终结符。

11.6.2 用于平衡括号串的递归下降分析器

现在来考虑一个扩展过的例子，看看如何为图11-25所示文法中的语法分类<*B*>设计递归函数*B*。在某个输入位置被调用的函数*B*，会销毁从那个位置开始的某个平衡括号串，并把输入光标留在紧临该平衡括号串之后的那个位置。

难点在于，要满足确定<*B*>这一目标，到底是使用可以立即成功的产生式(1)，<*B*> $\to \epsilon$，还是使用产生式(2)，也就是<*B*> →(<*B*>) <*B*>。而我们要遵循的策略是，只要下一个终结符为(，就使用产生式(2)，只要下一个终结符是)或是端记号，就使用产生式(1)。

函数*B*如图11-27b所示，在它之前的图11-27a中是一些重要的辅助要素，这些元素包括以下几点。

(1) 常量`FAILED`被定义为函数*B*没能在输入中找到平衡括号串时的返回值。`FAILED`的值与`NULL`相同。后者的值也表示一棵空树。不过，如果*B*成功的话，它返回的分析树不会为空，所以`FAILED`的这一定义是不可能有二义性的。

(2) 类型`NODE`和`TREE`的定义。节点是由标号字段（字符），以及指向最左子节点和右兄弟节点的指针组成的。标号`'B'`表示标号为*B*的节点，`'('`和`')'`分别表示标号为左括号和右括号的节点，而`'e'`则表示标号为ϵ的节点。与5.3节中最左子节点右兄弟节点结构不同的是，这里为指向节点的指针选择的类型是`TREE`而非`pNODE`，因为这里的这些指针多用来作为树的表示。

(3) 下面要描述的3个辅助函数和函数*B*的原型声明。

(4) 两个全局变量。第一个是`parseTree`，存放着由对*B*的第一次调用返回的分析树。第二个是`nextTerminal`，它是输入游标，指向输入终结符串中的当前位置。请注意，`nextTerminal`具有全局性是很重要的，这样当*B*的一次调用返回时，输入游标所在的位置对执行这次调用的*B*的副本而言就是已知的。

(5) `main`函数。在这一简单的演示中，`main`将`nextTerminal`置为指向特定测试串`()()`开头的位置，而且调用*B*的解雇被放置在`parseTree`中。

(6) 3个辅助函数可以创建树节点，而且，如果需要的话，可以组合子树以形成更大的树。它们分别是

(a) *makeNode*0(x)函数创建的节点没有子节点，也就是说，它创建的是叶子节点，而且用符号x作为该叶子节点的标号。返回的是由这一个节点组成的树。

(b) *makeNode*1(x, t)函数创建的节点具有一个子节点。新节点的标号为x，而且其子节点是树t的根节点。返回的是根节点为所创建节点的树。请注意，`makeNode1`要利用`makeNode0`创建根节点，然后让树t的根节点成为所创建根节点的最左子节点。我们假设所有的最左子节点和右兄弟节点指针一开始都是`NULL`，而且它们就是，因为它们都是由`makeNode0`创建的，该函数显然将它们置为了`NULL`。因此，`makeNode1`并不一定要把`NULL`存储到树t根节点的`rightSibling`字段中，不过这样做是明智的安全之举。

(c) 函数*makeNode*4(x, t_1, t_2, t_3, t_4)创建的节点具有4个子节点。该节点的标号是x，而其子节点按照从左到右的次序分别是树t_1、t_2、t_3和t_4的根节点，返回的是用所创建节点作为根节点的树。请注意，`makeNode4`要利用`makeNode1`创建一个新的根节点，并将t_1附加到该节点上，然后用右兄弟节点指针把其余的树串联起来。

```
#define FAILED NULL

typedef struct NODE *TREE;
struct NODE {
    char label;
    TREE leftmostChild, rightSibling;
};

TREE makeNode0(char x);
TREE makeNode1(char x, TREE t);
TREE makeNode4(char x, TREE t1, TREE t2, TREE t3, TREE t4);
TREE B();

TREE parseTree; /* 存放分析的结果 */
char *nextTerminal; /* 输入字符串中的当前位置 */

void main()
{
    nextTerminal = "()()"; /* 在实际应用中,终结符串是从输入读取的 */

    parseTree = B();
}

TREE makeNode0(char x)
{
    TREE root;

    root = (TREE) malloc(sizeof(struct NODE));
    root->label = x;
    root->leftmostChild = NULL;
    root->rightSibling = NULL;
    return root;
}

TREE makeNode1(char x, TREE t)
{
    TREE root;

    root = makeNode0(x);
    root->leftmostChild = t;
    return root;
}

TREE makeNode4(char x, TREE t1, TREE t2, TREE t3, TREE t4)
{
    TREE root;

    root = makeNode1(x, t1);
    t1->rightSibling = t2;
    t2->rightSibling = t3;
    t3->rightSibling = t4;
    return root;
}
```

图11-27(a)　递归下降分析器的辅助函数

```
        TREE B()
        {
(1)         TREE firstB, secondB;
(2)         if(*nextTerminal == '(') /* 遵循产生式2 */ {
(3)             nextTerminal++;
(4)             firstB = B();
(5)             if(firstB != FAILED && *nextTerminal == ')') {
(6)                 nextTerminal++;
(7)                 secondB = B();
(8)                 if(secondB == FAILED)
(9)                     return FAILED;
                    else
(10)                    return makeNode4('B',
                            makeNode0('('),
                            firstB,
                            makeNode0(')'),
                            secondB);
                }
                else /* 对B的第一次调用失败了 */
(11)                return FAILED;
            }
            else /* 遵循产生式1 */
(12)            return makeNode1('B', makeNode0('e'));
        }
```

图11-27(b)　为平衡括号串构建分析树的函数

现在可以一行行考虑图11-27b所示的程序了。第(1)行是两个局部变量`firstB`和`secondB`的声明，这两个局部变量的作用是存放在尝试产生式(2)的情况下对B的两次调用所返回的分析树。第(2)行会测试输入的下一个终结符是否为`(`。如果是，我们就将在产生式(2)的右部中查找实例，如果不是，就要假设使用的是产生式(1)，而且ϵ就是该平衡串。

在第(3)行，我们要递增`nextTerminal`，因为当前输入`(`已经匹配上了产生式(2)右部中的`(`。我们现在已经让输入游标处在恰当的位置，它对应的对B的调用将为产生式(2)右部中的第一个<B>找到平衡串。对B的这次调用是在第(4)行发生的，而该调用返回的树被存储在变量`firstB`中，它随后会被装配成与当前对B的调用对应的分析树。

第(5)行要检查是否仍然有能力找到平衡串。也就是说，首先要确定第(4)行对B的调用没有失败。然后测试`nextTerminal`当前的值是否为`)`。回想一下，当B返回时，`nextTerminal`指向要用来形成平衡串的下一个输入终结符。如果要匹配产生式(2)的右部，而且已经匹配了`(`与第一个<B>，那么就必须匹配`)`，这就解释了该测试的第二部分。只要该测试的任何一部分失败，当前对B的调用就会在第(11)行失败。

若通过了第(5)行的测试，则在第(6)和第(7)行要把输入游标移过刚发现的右括号，并再次调用B，以匹配产生式(2)中的最后一个<B>。返回的树被临时存储在`secondB`中。

如果第(7)行对B的调用失败，`secondB`的值就会是`FAILED`。第(8)行会检测这种情况，而且当前对B的调用也会失败。

第(10)行代表的是成功找到平衡括号串的情况。我们要返回由`makeNode4`构建的树。该树具有标号为'`B`'的根节点以及4个孩子。第一个是标号为`(`的叶子节点，它是由`makeNode0`构造的。第二个是存储在`firstB`中的树，它是通过第(4)行对B的调用产生的分析树。第三个

是标号为)的叶子节点，第四个则是由第(7)行对B的第二次调用返回的分析树，它存储在secondB中。

只有在第(5)行的测试失败时，才会使用第(11)行。第(12)行处理的是第(1)行的初始测试没能在第一个字符的位置找到(的情况。在这种情况下，假设产生式(1)是正确的。该产生式的右部为ϵ，因此我们没有销毁任何输入，但返回了一个由makeNode1创建的节点，其标号为B而且有一个标号为ϵ的子节点。

✦ 示例 11.9

假设在输入中有终结符串()()ENDM。这里的ENDM代表字符'\0'，它是在C语言中用来标记字符串结尾的。图11-27a中main函数对B的调用确定了(是当前的输入，而且第(2)行的测试会成功。因此，nextTerminal在第(3)行会前移，而且第(4)行会进行对B的第二次调用，表示为图11-28中的“调用2”。

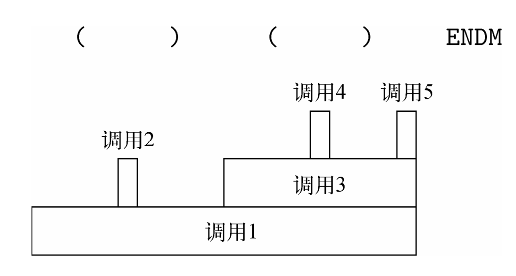


图11-28 在处理输入()()ENDM时进行的调用

在调用2中，第(2)行的测试失败，因此在第(12)行会返回图11-29a所示的树。现在回到调用1，其中在第(5)行时，nextTerminal指向的是)，而且图11-29a的树在firstB中。因此，第(5)行的测试会成功。我们在第(6)行前移nextTerminal，并在第(7)行调用B。这是图11-28中所示的“调用3”。

在调用3中，我们在第(2)行成功，在第(3)行前移nextTerminal，并在第(4)行调用B，该调用就是图11-28中的“调用4”。就和调用2一样，调用4也会在第(2)行的测试失败，并在第(12)行返回一棵类似图11-29a的树但有所不同。

我们现在回到了调用3，其中nextTerminal仍然指向)，firstB（是此次对B的调用的局部变量）存放着一棵类似图11-29a这样的树，而且有着第(5)行的控制。这次测试会成功，而且我们会在第(6)行前移nextTerminal，所以它现在指向的是ENDM。我们在第(7)行进行对B的第五次调用。该调用在第(2)行的测试会失败，并在第(12)行返回图11-29a的另一个副本。这棵树称为对应调用3的secondB的值，并且第(8)行的测试也失败了。因此，在调用3的第(10)行，我们要构建如图11-29b所示的树。

至此，调用3在第(8)行成功地返回到调用1，这时调用1的secondB存放着图11-29b中的树。就像在调用3中那样，第(8)行的测试会失败，而且我们在第(10)行要构建一棵有着新根节点的树，其第二个孩子是图11-29a所示树的一个副本（这棵树被存放在调用1的firstB中），而且它的第四个孩子是图11-29b中的树。得到的树被main函数放置在parseTree中，它的样子如图11-29c所示。

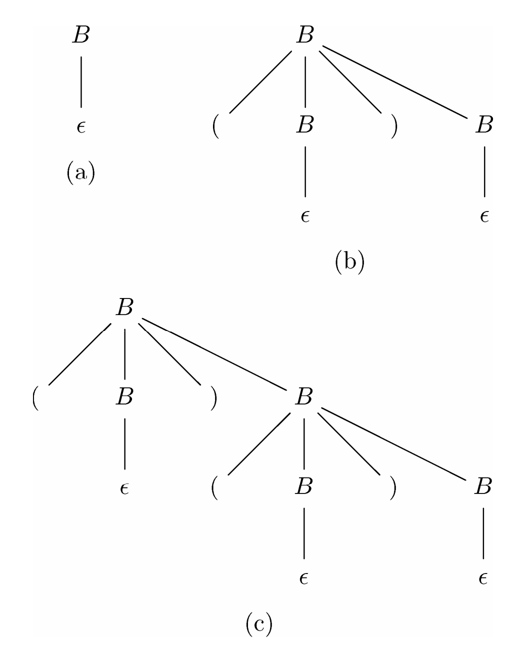


图11-29　由对B的递归调用构建的树

11.6.3　递归下降分析器的构建

虽然不能针对所有文法，但可以将图11-27中用到的技术扩展到适用于很多文法。关键要求是，对各语法分类<*S*>，如果存在不止一个以<*S*>为左部的产生式，那么通过查看当前唯一的终结符（通常被称为前瞻符号），就可以确定那个需要得到尝试的以<*S*>为左部的产生式。例如，在图11-27中，我们的决策就是，只要前瞻符号是`(`，就选取右部为(<*B*>) <*B*>第二个产生式，而要是前瞻符号为`)`或`ENDM`，就选定右部为ϵ的第一个表达式。

一般来说，我们不可能弄清对某定义文法而言是否存在总能做出正确决定的算法。对图11-27来说，我们声明所陈述的策略是行得通的，但并未对此加以证明。不过，如果拥有自己相信行得通的决策，那么就可以按照如下方式为各语法分类<*S*>设计函数*S*。

(1) 检查前瞻符号，并决定要尝试哪个产生式。假设被选中的产生式右部为$X_1X_2\cdots X_n$。

(2) 对$i = 1$、2、…、n，为X_i进行以下操作。

(a) 如果X_i是终结符，检查前瞻符号是否为X_i。如果是，则前移输入游标。如果不是，那么这次对*S*的调用就失败了。

(b) 如果X_i是语法分类，比方说是<*T*>，就调用对应该语法分类的函数*T*。如果*T*以失败状态返回，就说明对*S*的调用失败了。如果*T*成功返回，就把返回的树存储起来以待随后使用。

如果在考虑完所有的X_i之后都没有失败，就创建各孩子按次序分别对应X_1、X_2、$\cdots X_n$的新节点，以组成一棵要返回的分析树。如果X_i是终结符，那么X_i的孩子就是新创建的以X_i为标号的叶子节点。如果X_i是语法分类，那么X_i的子节点就是在与X_i对应的函数完成调用时返回的树的根节点。图11-29就是这种树构建过程的示例。

如果语法分类<*S*>表示所含字符串有待识别和分析的语言，就要在第一个输入的终结符处

放置输入游标，开始分析过程。如果输入在语言L(<S>)中，对函数S的调用就会使得对应该输入的分析树被构建起来，而如果不在的话，对S的调用就会返回失败。

11.6.4　习题

(1) 给出针对以下输入，图11-27所示的程序所执行的调用序列，其中每种情况下最后都跟着端记号ENDM符号。

(a) (( ))

(b) (( ) ( ))

(c) ( )) (

(2) 考虑以下表示数字的文法。

<数字> → <数码> <数字> | ϵ

<数码> → 0 | 1 | ⋯ | 9

设计对应该文法的递归下降分析器，也就是说，编写两个函数，其中一个对应<数字>，另一个对应<数码>。大家可以遵照图11-27中的格式，并假设存在makeNode1这种根节点具有指定数目子节点的树。

(3) ** 假设把习题(2)中对应<数字>的生成式写为

<数字> →<数码> <数字> | <数码>

或

<数字> →<数字> <数码> | ϵ

是否还能够设计递归下降分析器？为什么？

(1)	$<L>$	→	($<E>$ $<T>$
(2)	$<T>$	→	, $<E>$ $<T>$
(3)	$<T>$	→	)
(4)	$<E>$	→	$<L>$
(5)	$<E>$	→	原子

图11-30　对应表结构的文法

(4) * 图11-30所示的文法定义了非空表，表的元素是由逗号分隔并由括号包围的。元素可以是原子或表结构体。在这里，<E>代表元素，<L>表示表，而<T>则对应“尾部”，也就是，要么是闭合的)，要么是由逗号和以)结尾的元素构成的配对。为图11-30中的文法编写递归下降分析器。

11.7　表驱动分析算法

正如我们在6.7节中看到过的，递归函数调用通常是用活动记录栈实现的。因为递归下降分析器中的函数完成的工作非常具体，所以可以用一个检查表并操作栈的函数来代替这些函数。

要记得，对应语法分类<S>的函数S首先要决定使用哪个产生式，然后经过一系列步骤，每个步骤都对应着所选产生式右部中的一个符号。因此，可以维持一个大致与活动记录栈对应的文法符号栈，而符号和语法分类都被放置在该栈中。当语法分类<S>位于栈顶时，首先要确定正确的产生式。然后用所选产生式的右部替换<S>，其中右部的左端位于栈顶。如果是终结符位于栈顶，就要确定它是否与当前输入符号匹配。如果是，我们就将其弹出栈并前移输入游标。

要从直觉上了解这种安排为何起作用，先假设递归下降分析器刚调用过对应语法分类<S>的函数S，而且选定的产生式右部为a$< B >< C >$。那么对应S的这一活动记录会在以下4个时候

处于活动状态。

(1) 在检验**a**是否在输入中时；

(2) 在进行对*B*的调用时；

(3) 在该调用返回而且*C*被调用时；

(4) 在对*C*的调用返回而且*S*完成调用时。

如果我们在表驱动分析器中直接用右部的符号（本例中是**a**<*B*><*C*>）替换<*S*>，那么该活动记录栈会在控制权返回对应递归下降分析器中*S*的活动时的输入位置曝光这些符号。

(1) 第一次曝光的是**a**，而且我们会检测**a**是否在输入中，就像函数*S*所做的那样。

(2) 第二次，紧接第一次之后发生，*S*会调用*B*，而<*B*>会位于栈顶，这会造成相同的行为。

(3) 第三次，*S*调用*C*，不过这里是<*C*>在栈顶，而且完成的是相同的工作。

(4) 第四次，*S*返回，而且我们不会发现更多替代<*S*>的符号。因此，活动记录栈中此点以下的符号之前存放在<*S*>中，但现在被暴露在外了。类似地，在*S*的活动记录以下的活动记录在递归下降分析器中会得到控制权。

11.7.1　分析表

如果不想写一系列的递归函数，也可以构建分析表（parsing table），它的每一行都对应着语法分类，每一列对应着可能的前瞻符号。在表示语法分类<*S*>的那一行中，对应前瞻符号*X*的项是前瞻符号为*X*时展开<*S*>必须用到的以<*S*>为左部的产生式的编号。

分析表中的某些项可能为空。假设需要展开的语法分类<*S*>，而且前瞻符号为*X*，但我们发现表示<*S*>的那行中对应*X*的那一项为空，就说明分析已经失败了。这种情况下，可以确定该输入不在此语言中。

✦　示例 11.10

图11-31表示了对应图11-25所示平衡括号串无二义文法的分析表。该分析表相当简单，因为其中只有一个语法分类。该表所表示的策略与11.6节中的示例所采用的策略是相同的。如果前瞻符号是(，那么展开时用到的是产生式(2)，也就是<*B*> → (<*B*>) <B>，否则展开时就要借助产生式(1)，或者说<*B*> → ϵ 。我们很快就会看到这样的分析表是如何使用的。

	(	)	ENDM
<*B*>	2	1	1

图11-31　对应平衡括号串文法的分析表

✦　示例 11.11

图11-32所示的是另一个分析表，它对应着图11-33所示的文法，该文法是图11-6所示语句文法的一个变种。

	w	c	{	}	s	;	ENDM
<*S*>	1		2		3		
<*T*>	4		4	5	4		

图11-32　对应图11-33所示文法的分析表

图11-33中的文法之所以具有这种形式，是为了可以用递归下降（或者等价地，用这里描述的表驱动分析算法）进行分析。要知道为什么这种形式是必要的，首先考虑一下图11-6所示文法中对应<*L*>的产生式。

<*L*> →<*L*> <*S*>| ϵ

如果当前的输入是开始语句的**s**这样的终结符，那么可知<*L*>一定至少由右部为<*L*> <*S*>第一个生成式展开一次。不过，如果不检查接下来的输入并弄清语句列中共有多少条语句，就没法确定要进行多少次展开。

(1)　<*S*> → **w c**<*S*>
(2)　<*S*> → **{** <*T*>
(3)　<*S*> → **s ;**

(4)　<*T*> → <*S*><*T*>
(5)　<*T*> → **}**

图11-33　可进行递归下降分析的、表示简单语句的文法

我们在图11-33中用到的方法是，记住程序块是由左花括号后面跟上0条或更多语句以及右花括号组成的。我们把这0条或更多语句以及右花括号称为“尾部”（tail），并用语法分类<*T*>表示它。图11-33中的产生式(2)说明，语句可以由左花括号后面加上尾部构成。产生式(4)和产生式(5)则表示尾部要么是语句后面跟上尾部，要么直接就是个右花括号。

决定用产生式(4)还是产生式(5)展开<*T*>是件非常简单的事。只有在右花括号是当前输入时，产生式(5)才行得通，而产生式(4)只在当前输入可以开始语句时才有效。在我们的简单文法中，开始语句的终结符只有**w**、**{**和**s**。因此，在图11-32中可以看到，在对应语法分类<*T*>的那行中，为这3个前瞻符号选择了产生式(4)，而为前瞻符号**}**选择了产生式(5)。对其他的前瞻符号而言，我们不可能用它们作为尾部的开头，所以就要在对应<*T*>的那行中把对应其他前瞻符号的位置留空。

同样，语法分类<*S*>的决定也很简单。如果前瞻符号为**w**，那么只有产生式(1)能起作用。如果前瞻符号为**{**，产生式(2)就是唯一可行的选择。而对前瞻符号**s**来说，只有产生式(3)是可行的。对其他前瞻符号来说，相应的输入是没法形成语句的。这些结论解释了图11-32中对应<*S*>的那一行。

11.7.2　表驱动分析器的工作原理

所有的分析表都可以被实质上的同一程序用作数据。这一*驱动器*程序具有同时存放着终结符和文法分类的文法符号栈。该栈可以被视作剩下的输入必须满足的目标，这些目标一定是按照从栈顶到栈底的次序得到满足的。

(1) 通过确定某终结符是输入的前瞻符号，可以满足终结符的目标。也就是说，只要终结符**x**在栈顶位置，就要检查前瞻符号是否为**x**，如果是，就从栈中弹出**x**，并读取要成为新前瞻符号的下一个输入终结符。

(2) 通过查询分析表中行对应<*S*>且列对应前瞻符号的项，可以满足语法分类目标<*S*>。

(a) 如果相应的项为空，那么就不能为该输入得出分析树，这样驱动器程序就失败了。

(b) 如果相应的项含有产生式*i*，就要把<*S*>从栈顶位置弹出，并把产生式*i*右部中的各个符号压入栈中。右部中的符号是按照从右至左的顺序被压入栈的，这样一来，右部的第一个符号最终就会处在栈顶的位置，而第二个符号就紧邻其下，以此类推。作

为特例，如果右部为ϵ，就只要把<*S*>从栈中弹出即可。

假设想确定字符串*s*是否在*L*(<*S*>)中。在这种情况下，要用输入中的*s* `ENDM`字符串[①]启动驱动器，并读取第一个终结符作为前瞻符号。活动记录栈一开始只由语法分类<*S*>组成。

✦ 示例 11.12

我们对输入`{w c s ; s ; }ENDM`使用图11-32中的分析表。图11-34展示了表驱动分析器所执行的处理步骤。表中所示栈的内容是按照栈顶内容位于最左侧的方式排列的，这样一来，当我们把栈顶位置的语法分类替换为它某一产生式的右部时，该右部就会出现在栈顶的位置，其中的符号都是按照正常次序排列。

	栈	前瞻符号	剩余输入
1)	<*S*>	{	`wcs;s;}ENDM`
2)	{<*T*>	{	`wcs;s;}ENDM`
3)	<*T*>	w	`cs;s;}ENDM`
4)	<*S*><*T*>	w	`cs;s;}ENDM`
5)	*wc*<*S*><*T*>	w	`cs;s;}ENDM`
6)	*c*<*S*><*T*>	c	`s;s;}ENDM`
7)	<*S*><*T*>	s	`;s;}ENDM`
8)	*s*;<*T*>	s	`;s;}ENDM`
9)	;<*T*>	;	`s;}ENDM`
10)	<*T*>	s	`;}ENDM`
11)	<*S*><*T*>	s	`;}ENDM`
12)	*s*;<*T*>	s	`;}ENDM`
13)	;<*T*>	;	`}ENDM`
14)	<*T*>	}	`ENDM`
15)	}	}	`ENDM`
16)	ϵ	ENDM	ϵ

图11-34　使用图11-32所示表格的表驱动分析器的处理步骤

图11-34中的第(1)行展示了初始情况。因为要测试字符串`{wcs;s;}`是否属于语法分类<*S*>，所以一开始活动记录栈中只存放着<*S*>。给定字符串的第一个符号`{`是前瞻符号，而且字符串的其余部分跟上`ENDM`就构成了剩下的输入。

如果查看图11-32中对应语法分类<*S*>和前瞻符号`{`的项，就知道必须按照产生式(2)展开<*S*>。该产生式的右部是{<*T*>，而且在我们到达第(2)行时，可以看到这两个文法符号已经替换了栈顶的<*S*>。

现在栈顶位置是终结符`{`。因此要将其与前瞻符号加以比较。因为栈顶和前瞻符号相符，所以我们要弹出栈顶内容，并将输入游标前移到下一个输入符号`w`，这样它就成了新的前瞻符号。这些改变反映在第(3)行中。

---

① 有时候端记号`ENDM`符号也是必要的，它可以作为告知我们已经到达输入末端的前瞻符号，其他时候它只是用来捕捉错误。例如，在图11-31中`ENDM`是必要的，因为我们在平衡括号串之后总有更多的括号，但在图11-32中它不是必要的，对应`ENDM`的那列中没有任何项就证明了一切。

接下来，<*T*>位于栈顶而且w是前瞻符号，查阅图11-32可知恰当的行动是用产生式(4)展开。因此将<*T*>从栈中弹出，并压入<*S*> <*T*>，如第(4)行中所见。同样，现在处于栈顶的<*S*>会被产生式(1)的右部替代，因为这是由图11-32中对应<S>的行与对应前瞻符号w的列决定的，这一改变反映在第(5)行中。在第(5)行和第(6)行之后，栈顶的终结符会与当前的前瞻符号相比较，因为每一对都能匹配，所以它们被弹出，而且输入游标前移。

这里要遵照第(7)到第(16)行，核实每一步都是根据分析表可以采取的合适行为。因为在各终结符到达栈顶时会与当时的当前前瞻符号匹配，所以我们不会失败。因此，字符串`{wcs;s;}`在语法分类<*S*>中，也就是说，它是语句。

11.7.3　分析树的构建

上面描述的算法可以分辨给定字符串是否在给定语法分类中，不过它并不会生成分析树。不过，对该算法进行简单的修改，它就能在输入字符串在初始化活动记录栈所用的语法分类中时给出相应的分析树。11.6节中描述过的递归下降分析器是从下向上构建分析树的，也就是说，从叶子节点开始，并在函数调用返回时逐渐将其组合成更大的子树。

而对表驱动分析器来说，自上而下地构建分析树要更方便。也就是说，我们从根节点开始，并且随着我们不断选择用来展开栈顶位置语法分类的产生式，就同时为构建中的分析树的节点创建了子节点，这些子节点对应着所选产生式右部中的符号。构建分析树的规则如下所述。

(1) 一开始，活动记录栈中只含有某个语法分类，比方说是<*S*>。我们将分析树初始化为只含一个标号为<*S*>的节点。栈中的<*S*>对应着正在构建的分析树中的一个节点。

(2) 一般情况下，如果活动记录栈含有符号 $X_1X_2\cdots X_n$ ，而且X_1在栈顶，那么当前分析树的叶子节点标号从左到右排列可以组合成以 $X_1X_2\cdots X_n$ 为后缀的字符串*s*。分析树的后*n*个叶子节点对应着栈中的符号，所以每个栈符号X_i都与标号为X_i的叶子节点对应。

(3) 假设语法分类<*S*>位于栈顶，而且选择用产生式<*S*>→ $Y_1Y_2\cdots Y_n$ 的右部替代<*S*>。我们会找到分析树中对应这一<*S*>的叶子节点（它是以语法分类为标号的最左子节点），并给它*n*个从左至右标号分别为Y_1、Y_2、…、Y_n 的子节点。而在右部为ϵ 的特例中，我们会创建一个标号为ϵ 的子节点。

✦　示例 11.13

我们按照图11-34中的步骤进行处理，并在这一过程中构建分析树。首先，在第(1)行，活动记录栈只由<*S*>组成，而且对应的树是如图11-35a所示的单个节点。在第(2)行要用产生式<*S*>→`{`<*T*>展开<*S*>，因此就为图11-35a中的叶子节点赋予了两个子节点，从左至右分别以`{`和<*T*>为标号。对应第(2)行的树如图11-35b所示。

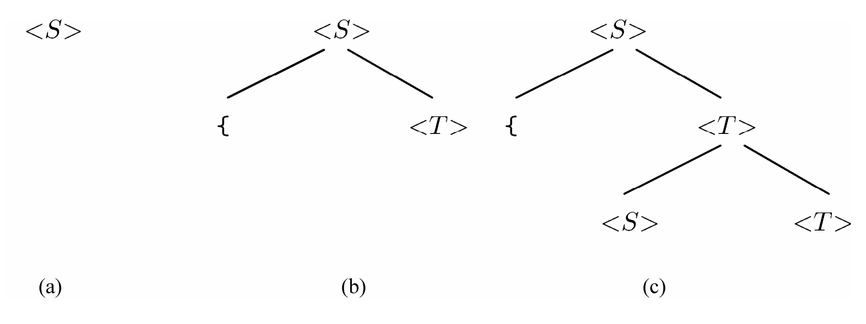


图11-35　构建分析树的前几步

第(3)行不会对分析树带来改变，因为我们匹配的是终结符，不会展开语法分类。不过在第(4)行要将<*T*>展开为<*S*> <*T*>，所以要为图11-35b中标号为<*T*>的叶子节点添加两个以这些符号为标号的子节点，如图11-35c所示。然后在第(5)行<*S*>被展开为**wc** <*S*>，这会让图11-35c中标号为<*S*>的叶子节点得到3个子节点。大家可以自行继续这一过程。最终的分析树如图11-36所示。

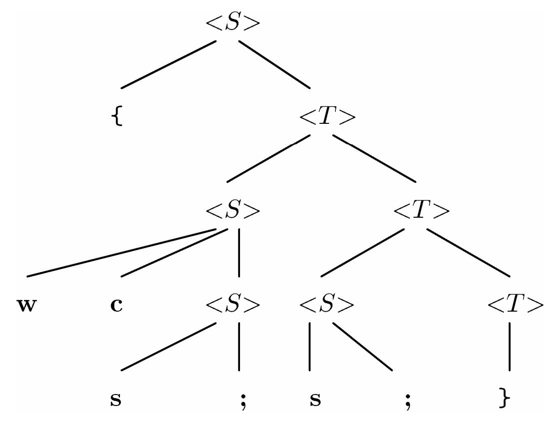


图11-36　对应图11-34中分析过程的完整分析树

11.7.4　让文法变得可分析

正如我们已经看到的，很多文法需要进行一些修改才能用我们在11.6和11.7这两节中了解的递归下降或表驱动方法进行分析。尽管无法保证任何文法都能修改为可用这两种方法分析，但有一些值得了解的小窍门，它们可以让这种使文法变得可分析的修改工作变得更高效。

第一个窍门是*消除左递归*。我们已经在示例11.11中指出过，产生式<*L*> → <*L*> <*S*>| ϵ 是不能用这些方法分析的，因为没办法弄清应用第一个产生式的次数。一般来说，只要对应某语法分类<*X*>的产生式的右部以<*X*>自身开头，我们就不清楚为了展开<*X*>要应用该产生式多少次。这种情况就叫作*左递归*。不过，通常可以重新排列有问题产生式的右部中的符号，使得<*X*>排在靠后的位置。这一步骤就叫作*左递归的消除*。

✦　示例 11.14

在之前讨论过的示例11.11中，我们看到<*L*>表示0个或更多<*S*>。因此可以通过调换<*S*>和<*L*>的位置消除左递归，这样一来就有

<*L*> → <*S*> <*L*>| ϵ

再举个例子，考虑一下表示数字的产生式：

<*数字*> → <*数字*> <*数码*> | <*数码*>

给定数码作为前瞻符号，就不知道要展开<*数字*>需要使用第一个产生式多少次。不过，我们看到数字是一个或多个数码，这样就可以重新排列第一个产生式的右部，得到

<*数字*> → <*数码*> <*数字*> | <*数码*>

这一对产生式就消除了左递归。

不过，示例11.14中的产生式还是不能用我们提过的方法分析。要让它们变得可分析，就需要用到第二个窍门——*提取左因子*（left factoring）。当对应语法分类<*X*>的两个产生式具有以相同符号*C*开头的右部时，只要前瞻符号是来自共有符号*C*的，我们就没法弄清该使用哪个产生式。

要为这些产生式提取左因子，就要创建表示这些产生式“尾部”，即表示右部中C之后的部分的语法分类<*T*>。然后把这两个对应<*X*>的产生式替换为<*X*>→C<*T*>和两个以<*T*>为左部的两个产生式。这两个产生式的右部是之前所说的“尾部”，也就是对应<*X*>的两个产生式中C之后的内容。

✦ 示例 11.15

考虑一下示例11.14中设计的对应<数字>的产生式：

<数字> → <数码> <数字> | <数码>

这两个产生式是以相同符号<数码>开始的。因此，当前瞻符号为数码时，我们没法分辨要使用哪个产生式。不过，如果提取左因子，得到

<数字> → <数码> <尾部>

<尾部> → <数字>| ϵ

就可以服从这些决定了。在这里，对应<尾部>的这两个产生式让我们选择对应<数字>的第一个产生式的尾部，也就是<数字>本身，或者第二个对应<数字>的产生式的尾部，即ϵ 。

现在，当必须扩展<数字>，并以数码作为前瞻符号时，就会用<数码> <尾部>替换<数字>，匹配该数码，然后可以根据数码后面跟着的内容选择如何展开尾部。也就是说，如果后面跟着另一个数码，那么就用<尾部>的第一个选择展开，而如果后面跟着的内容不是数码，则可知已经看到整个数字，并用ϵ 来替换<尾部>。

11.7.5　习题

(1) 为下列输入字符串模拟使用图11-32所示分析表的表驱动分析器。

(a) `{s;}`
(b) `wc{s;s;}`
(c) `{{s;s;}s;}`
(d) `{s;s}`

(2) 对习题(1)中取得成功的那些分析，给出分析过程中构建分析树的过程。

(3) 利用图11-31中的分析表，对11.6节习题(1)中的输入串模拟表驱动分析器。

(4) 给出习题(3)的分析过程中分析树的构建情况。

(5) * 如下文法

(a) <语句> → **if**（条件）
(b) <语句> → **if**（条件）<语句>
(c) <语句> → 简单语句

表示C语言中的选择语句。它是没法用递归下降分析器和表驱动分析器进行分析的，因为如果有前瞻符号**if**，就没办法确定要使用前两个产生式中的哪一个。为该文法提取左因子，使它可以用11.6节和本节介绍的算法分析。提示：在提取左因子时，就得到了具有两个产生式的新语法分类。一个的右部为ϵ ，而另一个的右部则以**else**开头。显然，当**else**作为前瞻符号时，要选择第二个产生式。其他前瞻符号都不能让我们选择这一产生式。不过，如果看看有哪些前瞻符号让我们用右部为ϵ 的产生式展开，就会发现这些前瞻符号中也有**else**。不过，也可以强行规定，在前瞻符号为**else**时决不展开为ϵ 。该选择对应着“**else**匹配之前未匹配的**then**”这一规则，因此这是“正确的”选择。你可能想找到一个在前瞻符号为**else**且展开为ϵ 时还能让分析器完成分析的例子。不过你会发现，在任意一次这样的分析中，构建的分析树总会为**else**匹配“错误的”**then**。

(6) ** 如下文法

<结构体> → **struct{**<字段列>

<字段列> → 类型　字段名；<字段列>

<字段列> → ϵ

需要一些修改才可以用本节和11.6节介绍的方法进行分析。重写该文法，使其可由递归下降和表驱动的方法分析，并构建相应的分析表。

11.8 文法与正则表达式

文法和正则表达式都是用于描述语言的表示法。我们在第10章中已经看到，正则表达式表示法与确定自动机和非确定自动机表示法是等价的，因为可由这3种表示法描述的语言集合是相同的。文法是否有可能是另一种与我们已经见过的这些表示法都等价的表示法？

答案是“不可能”，因为文法要比我们在第10章中介绍的正则表达式之类的表示法更强大。这里要分两步展现文法的表现力。首先，我们将证实每种能用正则表达式描述的语言也都能用文法来描述。接着我们会给出一种可以由文法描述，但不能用正则表达式描述的语言。

11.8.1 用文法模拟正则表达式

这种模拟背后的直觉思路就是，正则表达式中的3种运算符（取并、串接和闭包）分别可以用一个或两个产生式“模拟”。正式地讲，可以通过对正则表达式R中出现的运算符的数量n进行完全归纳，证明如下命题。

命题。对每个正则表达式R来说，都存在某一文法，满足对文法中的语法分类$<S>$而言，有$L(<S>) = L<R>$

也就是说，由正则表达式表示的语言也是语法分类$<S>$的语言。

依据。依据情况是$n = 0$，也就是正则表达式R中未出现运算符的情况。R要么是单个符号，比方说$\mathbf{x}$，要么是ϵ或$\varnothing$。我们创建一个新的语法分类$<S>$。在$R = \mathbf{x}$的第一种情况下，还要创建产生式$<S> \to \mathbf{x}$。因此，$L(<S>) = \{\mathrm{x}\}$，而且$L<R>$也是相同的单字符串语言。如果R是ϵ，同样可以为$<S>$创建产生式$<S> \to \epsilon$，而如果$R = \varnothing$，则根本不用为$<S>$创建产生式。这样当R为ϵ时，$L(<S>)$是$\{\epsilon\}$；而当R是$\varnothing$时，$L(<S>)$是$\varnothing$。

归纳。假定归纳假设对具有不超过n个运算符的正则表达式成立。设R是其中出现$n+1$个运算符的正则表达式。总共有3种情况，具体取决于构建R所应用的最后一个运算符是取并、串接还是闭包。

(1) $R = R_1 \mid R_2$。因为这里有一个运算符（即取并运算符|）既不属于R_1也不属于R_2，所以可知R_1和R_2中运算符的个数都不可能超过n。因此，归纳假设适用于R_1和R_2，而且我们可以找到具有语法分类$<S_1>$的文法G_1，以及具有语法分类$<S_2>$的文法G_2，分别满足$L(<S_1>)=L(R_1)$和$L(<S_2>)=L(R_2)$。为了避免出现两个文法相互融合的巧合出现，我们可以假设，在构造新文法的过程中，所创建的语法分类的名称一直都不会在另一个文法中出现。这样一来，G_1和G_2中就不会有相同的语法分类。创建一个新的语法分类$<S>$，它既未出现在G_1和G_2中，也没有出现在为其他正则表达式构建的其他文法中。除了对应G_1和G_2的两个产生式外，我们还要添加两个产生式

$$<S> \to <S_1> \mid <S_2>$$

那么$<S>$的语言只由$<S_1>$和$<S_2>$的语言中所有的字符串组成。这两个语言分别是$L(R_1)$和$L(R_2)$，所以有

$$L(<S>) = L(R_1) \cup L(R_2) = L(R)$$

这正是我们想要的结果。

(2) $R = R_1R_2$。就像情况(1)那样，假设存在文法 G_1 和 G_2，它们分别具有语法分类$<S_1>$和$<S_2>$，满足$L(<S_1>)=L(R_1)$和 $L(<S_2>)=L(R_2)$。然后创建新的语法分类$<S>$，并在 G_1 和 G_2 产生式的基础之上添加产生式

$$<S> \rightarrow <S_1> <S_2>$$

然后就有$L(<S>) = L(<S_1>)\ L(<S_2>)$。

(3) $R = R_1^*$。设 G_1 是具有语法分类$<S_1>$的文法，满足$L(<S_1>) = L(R_1)$。创建新语法分类$<S>$，并添加两个产生式

$$<S> \rightarrow <S_1> <S> | \epsilon$$

因为$<S>$可生成由0个或更多$<S_1>$构成的串，所以有$L(<S>) = (L(<S_1>))^*$。

✦　示例 11.16

考虑正则表达式**a|bc***。首先要为该表达式中的3个符号创建语法分类。[①]因此，就得到产生式

$$<A> \rightarrow \mathbf{a}$$

$$<B> \rightarrow \mathbf{b}$$

$$<C> \rightarrow \mathbf{c}$$

根据正则表达式的组合规则，我们的表达式会被分组为**a|** (**b**(**c**)*)。因此，首先要创建对应**c***的文法。根据之前所述的规则(3)，我们要在产生式$<C> \rightarrow$**c**（就是对应正则表达式**c**的文法）的基础之上添加产生式

$$<D> \rightarrow <C> <D> | \epsilon$$

这里的语法分类$<D>$是随意选择的，可以是除了已经被使用过的$<A>$、$<B>$和$<C>$之外的任何语法分类。要注意到

$$L(<D>) = (L(<C>))^* = c^*$$

现在我们需要对应**bc***的文法。可以取只由产生式$<B> \rightarrow$**b**组成的对应**b**的文法，以及对应**c***的文法，即

$$<C> \rightarrow \mathbf{c}$$

$$<D> \rightarrow <C> <D> | \epsilon$$

我们要创建新的语法分类$<E>$，并添加产生式

$$<E> \rightarrow <B> <D>$$

之所以使用该产生式，是因为之前提到的对应串接情况的规则(2)。它的右部包含$<B>$和$<D>$，因为它们分别是对应正则表达式**b**和**c***的语法分类。因此对应**bc***的文法是

$$<E> \rightarrow <B> <D>$$

$$<D> \rightarrow <C> <D> | \epsilon$$

$$<B> \rightarrow \mathbf{b}$$

$$<C> \rightarrow \mathbf{c}$$

而且语法分类$<E>$的语言就是所需的。

最后，要得到对应整个正则表达式的文法，就要用到对应取并运算的规则(1)。这要引入新的语法分类$<F>$，以及产生式

① 如果这些符号出现两次或多次，并不需要为符号的每次出现创建新的语法分类，只需要为每种符号创建一个语法分类就足够了。

<*F*> → <*A*>|<*E*>

请注意，语法分类<*A*>对应子表达式**a**，而<*E*>则对应子表达式**bc***。得到的文法就是

<*F*> → <*A*>|<*E*>
<*E*> → <*B*> <*D*>
<*D*> → <*C*> <*D*>| ϵ
<*A*> → **a**
<*B*> → **b**
<*C*> → **c**

而语法分类<*F*>的语言就是给定正则表达式所表示的语言。

11.8.2 有文法但没有正则表达式的语言

现在要证实文法并非只有正则表达式那么强大。我们要通过展示一种只有文法但没有正则表达式的语言来做到这一点。我们将这一语言称为E，它是由一个或更多0后面跟上相等数量的1组成的字符串的集合。也就是说

$$E = \{01，0011，000111，...\}$$

要描述E中的字符串，有一种基于指数的实用表示方法。设s^n（其中s是字符串而n是整数）代表$ss\cdots s$（n个s），也就是说，s与它自身串接n次。那么

$$E = \{0^1 1^1, 0^2 1^2, 0^3 1^3, ...\}$$

或者使用集合形成法表示就是

$$E = \{0^n 1^n | n \geqslant 1\}$$

首先，我们要相信可以用文法描述E。以下文法就可以完成这一工作。

(1) <*S*> → 0 <*S*> **1**

(2) <*S*> → 01

大家可以使用依据产生式(2)说明01在L(<*S*>)中。在第二轮中，我们可以使用产生式(1)，用01替换右部中的<*S*>，这样就为L(<*S*>)得到了$0^2 1^2$。再一次应用产生式(1)，用$0^2 1^2$替换右部中的<*S*>，就说明$0^3 1^3$也在L(<*S*>)中，等等。一般来说，$0^n 1^n$要求使用产生式(2)一次，并随后使用产生式(1) $n-1$次。因为我们用这两个产生式不能产生别的字符串，所以可知$E = L$(<*S*>)。

11.8.3 证明E不能用任何正则表达式定义

现在要证明E不能用正则表达式描述。这里证明E不能用任何确定有限自动机描述要更容易些。这一证明过程也能证明E没有正则表达式，因为如果E是正则表达式R的语言，我们就可以利用10.8节中的技巧将R转换成等价的确定有限自动机。该确定有限自动机就定义了语言E。

因此，假设E是某确定有限自动机A的语言。那么A就会有若干数量的状态，比方说是m个状态。考虑一下当A接收输入000⋯时会发生什么。我们这里把该未知自动机A的初始状态叫作s_0。A一定有针对输入0的、从状态S_0到某个我们称之为S_1的状态的转换。从该状态出发，另一个0可以把A带到称为S_2的状态，等等。一般地说，在读入i个0后就处在状态S_0中，如图11-37所示。[①]

---

① 大家应该记住，我们其实并不知道A的状态的名称，而是只知道A具有m个状态（其中m为某整数）。因此，s_0、⋯、s_m并不是A中状态的真正名称，而只是我们为了方便称呼而为这些状态赋予的名称。这并不像看起来这么奇怪。打个比方，我们一般会创建数组s，用0到m作为索引，并在$s[i]$中存储某值，该值可以是自动机A的状态名称。然后可以在程序中用$s[i]$代指该状态，而不是用它本身的名称。



图11-37　为自动机A输送0

鸽巢原理

证明语言E没有确定有限自动机的过程要用到称作鸽巢原理（pigeonhole principle）的技巧，我们通常将该原理陈述为：

“如果m+1只鸽子飞进m个鸽巢，那么至少有一个巢中有两只鸽子。”

在这种情况下，鸽巢就相当于自动机A的状态，而鸽子就是A在看到0个、1个、2个直至m个0后所处的m个状态。

请注意，为了应用鸽巢原理，这里的m必须是有限的。7.11节中讲过的无限酒店的故事告诉我们，对立的命题对无限集来说是可以成立的。在那个例子中，我们看到一家有着无数个房间（对应鸽巢）的酒店，以及数量比房间数多1的客人（对应鸽子），不过还是有可能为每个客人分配一个房间，而不用把两个客人安排到同一房间中。

现在假设A刚好有m个状态，而且s_0、s_1、⋯、s_m总共有m+1个状态。因此不可能让这些状态全都不同。在0到m的范围中，肯定存在两个不同的整数i和j，使得s_i和s_j其实为相同状态。如果假设i是i和j之间的较小者，那么图11-37的路径之中一定至少存在一条环路，如图11-38所示。在实际应用中，可能存在比图11-38中更多的环路和状态重复。还要注意，i可以是0，在这种情况下，图11-38所示的从s_0到s_i的路径起始就是一个节点。同样，s_j可以是s_m，这种情况下从s_j到s_m的路径就只是个节点。

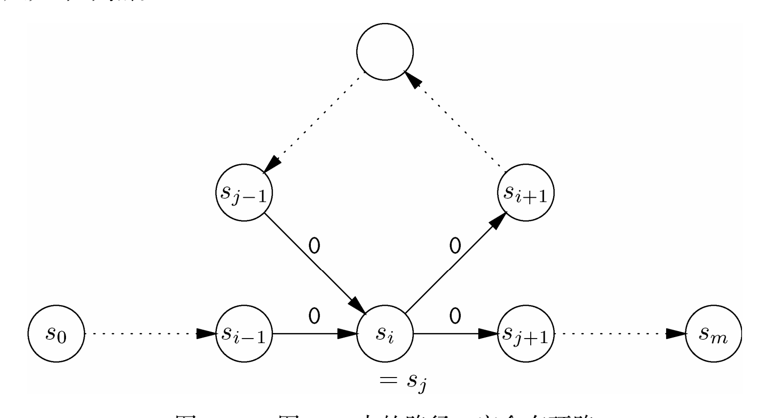


图11-38　图11-37中的路径一定含有环路

图11-38暗示了自动机A不能“记住”自己已经看到过多少个0。如果处在状态s_m中，它可能已经刚好看到了m个0，这样的话，如果我们从状态m开始，并为A提供刚好m个1，那么A一定会到达接受状态，如图11-39所示。

不过，假设我们为A提供了一个有m+j–i个0的串。看看图11-38，就会发现i个0可以将A从s_0带到与s_j相同的s_i。我们还会看到m–j个0会把A从状态s_j带到s_m。因此，m–j+i个0就可以把A从状态s_0带到s_m，如图11-39中靠上方的路径那样。

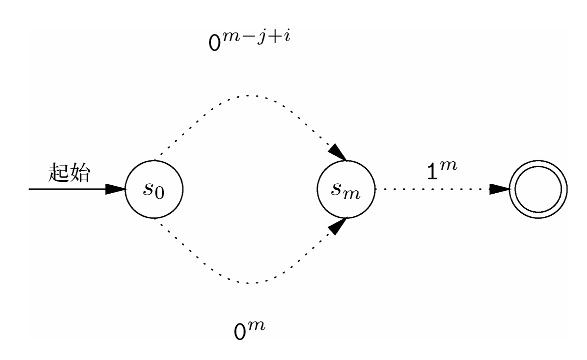


图11-39 自动机A不能区分它到底是看到了m个0还是m–j+1个0

因此，m–j+i个0后面跟上m个1也可以把A从 s_0 带到接受状态。换句话说，字符串 $0^{m-j+i}\,1^m$ 也在A的语言中。但因为j比i大，所以该串中的1要比0多，这样就不在语言E中。因此可得出A的语言并不是E的结论，正如我们所假设的那样。

因为一开始只假设了E具有确定有限自动机，而且最终推出了矛盾，所以可推断出假设不成立，也就是说，E没有确定有限自动机。因此，E也没有正则表达式。

语言$\{0^n1^n | n \geqslant 1\}$只是无数种可由文法指定但不能用正则表达式表示的语言中的一个例子。本节习题中还会给出另外一些例子。

11.8.4 习题

(1) 给出定义以下正则表达式语言的文法。

(a) **(a|b)*a**

(b) **a*|b*|(ab)***

(c) **a*b*c***

文法不能定义的语言

有人可能会问文法是否为描述语言的最强大表示法，答案是：“绝不可能！”我们可以证明一些简单语言并不具有文法，尽管证明技巧并不在本书要介绍的范围之内。这种的语言的一个例子就是由相同数量的0、1和2按次序构成的串形成的集合，也就是

{012，001122，000111222，…}

要举一个更强大的语言描述方法的例子，可以考虑一下C语言本身。对任意文法，以及它的任一语法分类<S>来说，都可以写出C语言程序以确定字符串是否在L(<S>)中。此外，确定字符串是否在上述语言中的C语言程序也不难编写。

但还是有C语言程序不能定义的语言。“不可判定问题”这一高雅理论就可用来证明某些问题是不能用任何计算机程序解决的。我们将在14.10节中简要讨论不可判定性以及一些不可判定问题的例子。

---

(2) * 证明平衡括号串集合不能由任何正则表达式定义。提示：这一证明过程与上述针对语言E的证明过程类似。假设平衡括号串集合具有含m个状态的确定有限自动机。为该自动机提供m个(，然后检查它进入的状态。证明该自动机可能被“愚弄”去接受某个不平衡的括号串。

(3) * 证明由形如0^n10^n的字符串（也就是两列等长的0由一个1分开）组成的语言是不可以用正则表达式定义的。

(4) * 大家有时候可能会看到一些谬误的断言，声称像本节中E这样的语言可以用正则表达式描述。推

理过程是，对各个n而言，$\mathbf{0}^n\mathbf{1}^n$ 就是定义只含 0^n1^n 这一个字符串的语言的正则表达式。因此下列正则表达式就是描述E的。

$\mathbf{01}|\mathbf{0}^2\mathbf{1}^2|\mathbf{0}^3\mathbf{1}^3|\cdots$

这一论证过程错在何处？

(5) * 另一个和语言有关的谬误论证声称E具有以下有限自动机。该自动机具有一个状态a，它既是起始状态又是接受状态。存在从a到它自身的针对符号0和1的转换。那么显然字符串 0^i1^i 能把自动机从状态a带到状态a，这样该字符串就被接受了。为什么这一论证过程没有证明E是某有限自动机的语言？

(6) ** 证明下列语言不能用正则表达式定义。

(a) $\{ww^R|w$是由a和b组成的字符串，w^R是与w排列次序相反的字符串$\}$

(b) $\{\ 0^i\ |\ i$是完全平方数$\}$

(c) $\{\ 0^i\ |\ i$是质数$\}$

其中哪些语言可以用文法定义？

11.9　小结

在阅读过本章之后，大家应该注意以下要点。

- （上下文无关）文法如何定义语言。
- 如何构建可以表示字符串文法结构的分析树。
- 二义文法有何歧义，为什么编程语言的规范中不需要二义文法。
- 可用来为某些类型的文法构建分析树的递归下降分析技术。
- 用于实现递归下降分析器的表驱动方式。
- 为什么文法与正则表达式或有限自动机相比是更强大的描述语言的表示法。

11.10　参考文献

上下文无关文法首先是由Chomsky [1956] 作为描述自然语言的范式研究的。同样的范式也被用来定义两种最早的重要编程语言的语法，这两种语言分别是Fortran (Backus et al. [1957])和Algol 60 (Naur [1963])。因此，上下文无关文法通常也称为巴科斯范式 (Backus-Naur Form，BNF)。Bar-Hillel, Perles and Shamir [1961]首先通过上下文无关文法的数学属性对其进行了研究。对上下文无关文法及其应用更为深入的研究见Hopcroft and Ullman [1979]或Aho, Sethi and Ullman [1986]。

递归下降分析器已经用于很多编译器和编译器编写系统中（见Lewis, Rosenkrantz and Stearns [1974]）。Knuth [1965]首先确定了LR文法，这是可以按照从左到右扫描输入的方式被确定分析的文法的最大自然分类。

Aho, A. V., R. Sethi, and J. D. Ullman [1986]. *Compiler Design: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass.

Backus, J. W. [1957]. "The FORTRAN automatic coding system," *Proc. AFIPS Western Joint Computer Conference*, pp. 188–198, Spartan Books, Baltimore.

Bar-Hillel, Y., M. Perles and E. Shamir [1961]. "On formal properties of simple phrase structure grammars," *Z. Phonetik, Sprachwissenschaft und Kommunikationsforschung* **14**, pp. 143–172.

Chomsky, N. [1956]. "Three models for the description of language," *IRE Trans. Information*

Theory **IT-2**:3, pp. 113–124.

Hopcroft, J. E. and J. D. Ullman [1979]. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass.

Knuth, D. E. [1965]. “On the translation of languages from left to right,” *Information and Control* **8**:6, pp. 607–639.

Lewis, P. M., D. J. Rosenkrantz, and R. E. Stearns [1974]. “Attributed translations,” *J. Computer and System Sciences* **9**:3, pp. 279–307.

Naur, P. (ed.) [1963]. “Revised report on the algorithmic language Algol 60,” *Comm. ACM* **6**:1, pp. 1–17.

第 12 章

命题逻辑

本章要介绍命题逻辑，是种最初目的是为了模型推理的代数，其历史要追溯到亚里士多德的时代。在更近的时代里，这种代数和很多代数一样，是实用的设计工具。例如，第13章就展示了命题逻辑是如何应用到计算机电路设计中的。逻辑的第三个用途是作为编程语言和系统（比如Prolog语言）的数据模型。很多通过计算机进行推理的系统，包括定理证明程序、程序验证程序，以及人工智能领域的应用，都是用基于逻辑的语言实现的。这些语言一般都利用了“谓词逻辑”，它扩充了命题逻辑的功能，是更为强大的逻辑形式。我们将在第14章中介绍谓词逻辑。

12.1　本章主要内容

12.2节从直观上说明了命题逻辑是什么，以及它为何实用。12.3节介绍了用于逻辑表达式的代数，它使用布尔值操作数，并且用到对布尔（真/假）值进行运算的`AND`、`OR`和`NOT`这样的逻辑运算符。这种代数通常称为*布尔代数*，是以首先将逻辑表达为代数的逻辑学家乔治·布尔的名字命名的。然后我们还会了解以下内容。

- 真值表是表示表达式逻辑意义的实用方式（12.4节）。
- 可以把真值表转换为对应相同逻辑函数的逻辑表达式（12.5节）。
- 卡诺图是一种用于简化逻辑表达式的实用制表技巧（12.6节）。
- 存在数量丰富的“重言式”，或可用于逻辑表达式的代数法则（12.7节和12.8节）。
- 命题逻辑的某些重言式让我们可以对“反证法”或“逆转命题法”这样的常用证明技巧加以解释（12.9节）。
- 命题逻辑也是经得起“演绎”的。所谓演绎，就是写出一行行内容，其中每一行要么是给定的，要么是能由之前的某些行来验证的（12.10节）。大多数人在学习平面几何时都了解过这种证明模式。
- 名为“分解”（resolution）的强大技巧有助于我们迅速作出证明（12.11节）。

12.2　什么是命题逻辑

山姆编写了如下含有`if`语句的C语言程序。

```
if (a < b || (a >= b && c == d)) ...
```
(12.1)

莎莉指出，该`if`语句中的条件表达式可以写为如下更简单的形式。

```
if (a < b || c == d) ...
```
(12.2)

莎莉是如何得出这一结论的？

她可能是按照如下方式推理的。假设a < b。那么参与OR运算的第一个条件在这两条语句中都为真，因此(12.1)和(12.2)这两条if语句中的then部分都会被接受。

现在假设a < b不成立。在这种情况下，只有当参与OR运算的第二个条件为真的情况下才会接受then部分。对语句(12.1)我们要询问

```
a >= b && c == d
```

是否为真。因为a < b为假，所以现在a >= b显然为真。因此，在(12.1)中，只有在c == d为真时才接受then部分。而对语句(12.2)，显然也是只有在c == d为真时才接受then部分。因此不管a、b、c、d的值分别是什么，要么是if语句都能带来接下来的then部分，要么是都不能。所以我们可以判断出莎莉是对的，简化过的条件表达式可以在不改变程序功能的情况下替换第一个表达式。

命题逻辑是让可以对逻辑表达式的真假进行推理的数学模型。我们将在12.3节中给出逻辑表达式的正式定义，不过现在可以把逻辑表达式视作对(12.1)和(12.2)这样的条件表达式的简化，这种简化抽象掉了C语言逻辑运算符求值次序的限制。

命题和真值

请注意，上述针对两个if语句的推理并不取决于a < b或相似的条件“意味着”什么。我们需要知道的只有条件a < b和a >= b是*互补的*，也就是说，当一个条件为真时另一个条件就为假，反之亦然。因此可以用符号p替换语句a < b，用表达式NOT p替代a >= b，并用符号q替换c == d。这里的符号p和q称为命题变量，因为它们可以代表任何“命题”，也就是任何可以具有某一*真值*（真或假）的语句。

逻辑表达式可以包含AND、OR和NOT这样的逻辑运算符。当逻辑表达式中逻辑运算符操作数的值已知时，表达式的值就可以通过下面这样的规则确定。

(1) 当且仅当p和q都为真时，p AND q为真，否则它为假。

(2) 当p为真，q为真，或两者都为真时，p OR q为真，否则它为假。

(3) 如果p为假，则NOT p为真，如果p为真，则它为假。

NOT运算符与C语言运算符!具有相同含义。运算符AND和OR分别类似于C语言的运算符&&和||，但存在技术差异。不过，只有在C语言表达式具有副作用时，这一细节才很重要。因为逻辑表达式的求值过程中是没有“副作用”的，所以可以把AND当作C语言运算符&&的同义词，把OR当作||的同义词。

例如，在(12.1)式中的条件可以写为逻辑表达式

$$p \text{ OR}\big((\text{NOT } p) \text{ AND } q\big)$$

而(12.2)式可以写为p OR q。我们对(12.1)和(12.2)这两个if语句的推理证明了如下一般性命题

$$p \text{ OR}\big((\text{NOT } p) \text{ AND } q\big) \equiv (p \text{ OR } q) \quad (12.3)$$

其中≡意味着“等价于”或“与……具有相同的布尔值”。也就是说，不管为命题变量p和q指定什么样的真值，≡的左边跟右边要么都为真，要么都为假。我们发现对上面的等价性而言，当p为真或当q为真时就都为真，而如果p和q都为假则为假。因此，我们得到了有效的等价。

因为p和q可以是任何命题，所以可以利用(12.3)式简化很多不同的表达式。例如，可以设p为

```
a == b+1 && c < d
```

而q为`a == c || b == c`。在这种情况下，(12.3)的左边就成了

```
(a == b+1 && c < d) ||
    ( !(a == b+1 && c < d) && (a == c || b == c))          (12.4)
```

请注意，我们为p和q的值加上了括号，以确保得到的表达式组合正确。

(12.3)式告诉我们(12.4)式可以简化为(12.3)式的右边，也就是

```
(a == b+1 && c < d) || (a == c || b == c)
```

再举个例子，设p是命题“天气晴朗”，而q是命题“乔伊带着伞”，那么(12.3)的左边就成了

“天气晴朗，否则不是晴天但乔伊带着伞。”

而右边也表示同样的事情，就是

“天气晴朗，否则乔伊带着伞。”

命题逻辑不能做什么

命题逻辑是一种实用的推理工具，但它是有局限性的，因为它不能洞悉命题内部并利用命题间的关系。比如，莎莉写出了if语句

```
if (a < b && a < c && b < c) ...
```

然后山姆指出只要写成

```
if (a < b && b < c) ...
```

就足够了。如果设p、q和r分别代表命题(`a < b`)、(`a < c`)和(`b < c`)，这样看起来山姆所说的就是

$$(p \text{ AND } q \text{ AND } r) \equiv (p \text{ AND } r)$$

不过这一等价性并不总是存在。例如，假设p和r为真，而q为假。那么右边就为真，而左边却为假。

山姆的简化结果是正确的，但这里并不是利用命题逻辑得到的。大家可能会回想起7.10节中介绍的<是一种具有传递性的关系。也就是说，只要p和r都为真，也就是`a < b`和`b < c`都成立，那么就有q为真，即`a < c`成立。

第14章将介绍一种叫作谓词逻辑的更为强大的模型，它允许我们为命题附加参数。这种特权让我们可以利用<这样的运算符的特殊属性。对我们来说，可以把谓词视作第7章和第8章的集合论中关系的名称。例如，我们可以创建谓词lt表示运算符<，并将p、q和r分别写为$lt(a, b)$、$lt(a, c)$和$lt(b, c)$。那么，根据表示lt属性的合适法则，比如传递性，就可以得到

$$\big(lt(a,b) \text{ AND } lt(a,c) \text{ AND } lt(b,c)\big) \equiv (lt(a,b) \text{ AND } lt(b,c))$$

其实，上式对任何满足传递律的谓词lt都是成立的，而不仅是对谓词<成立。

12.3　逻辑表达式

正如12.2节中提到的，逻辑表达式可以按照以下方式递归地定义。

依据。命题变量以及逻辑常量`TRUE`和`FALSE`都是逻辑表达式，这些都是原子操作数。

归纳。如果E和F是逻辑表达式，那么

(a) E AND F。如果E和F都为TRUE，则该表达式的值为TRUE，否则为FALSE。

(b) E OR F。如果E为TRUE、F为TRUE或两者都为TRUE，则该表达式的值为TRUE，如果E和F都为FALSE，则该表达式的值为FALSE。

(c) NOT E。若E为FALSE，则该表达式的值为TRUE，若E为TRUE则其值为FALSE。

也就是说，逻辑表达式可以用二元中缀表达式AND和OR，以及一元前缀表达式NOT构建。与其他代数一样，我们需要用括号进行组合，不过在某些情况下，也可以利用运算符的优先级和结合性消除多余的括号对，正如我们在涉及这些逻辑运算符的C语言条件表达式中所做的那样。在12.4节中，将看到出现在逻辑表达式之外的更多逻辑运算符。

✦ 示例 12.1

下面是一些逻辑表达式的例子

(1) TRUE

(2) TRUE OR FALSE

(3) NOT p

(4) p AND (q OR r)

(5) (q AND p) OR (NOT p)

在这些表达式中，p、q和r都是命题变量。

12.3.1 逻辑运算符的优先级

与其他类型的表达式一样，我们也可以为逻辑运算符指定优先级，而且可以利用这样的优先级消除某些括号对。我们已经看到的这些逻辑运算符的优先次序分别是NOT（最高），然后是AND，再是OR（最低）。虽然我们将会看到AND和OR具有结合性，怎样分组都无关紧要，但它们通常是从左组合的。而一元前缀运算符NOT只能从右起分组。

✦ 示例 12.2

NOT NOT p OR q被分组为(NOT (NOT p)) OR q。而NOT p OR q AND r被分组为(NOT p) OR (q AND r)。大家应该可以看出，AND、OR和NOT的优先级和结合性与算术运算符×、+和一元的–之间存在相似性。例如，本例所述的第二个表达式就可以类比为算术表达式$-p+q\times r$，它有着相同的分组方式，$(-p)+(q\times r)$。

12.3.2 逻辑表达式的求值

当逻辑表达式中的所有命题变量都被赋予真值时，表达式本身也得到一个真值。我们可以像为算术表达式或关系表达式求值那样，为逻辑表达式求值。

这一过程通过对应逻辑表达式的表达式树可以得到最好的体现。图12-1就是对应逻辑表达式p AND (q OR r) OR s的表达式树。给定某一真值赋值，也就是，为各变量赋值TRUE或FALSE，可以从表示原子操作数的叶子节点开始。每个原子操作数要么是逻辑常量TRUE或FALSE之一，要么是根据真值赋值给定了TRUE或FALSE之中某一个值的变量。然后向上处理该表达式树。一旦某内部节点v的子节点的值已知，就可以对这些值应用处在节点v的运算符，并为节点v产生真值。根节点的真值就是整个表达式的真值。

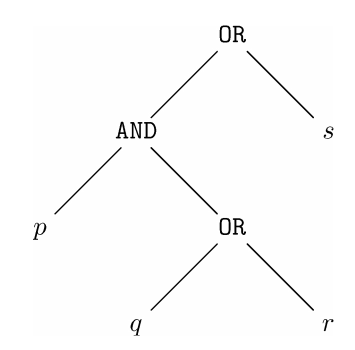


图12-1　表示逻辑表达式p AND (q OR r) OR s的表达式树

✦ 示例 12.3

假设要为表达式p AND (q OR r) OR s求值，其中p、q、r和s的真值赋值分别是TRUE、FALSE、TRUE和FALSE。我们首先要考虑图12-1中最低的内部节点，也就是表示表达式q OR r的内部节点。因为q为FALSE，而r为TRUE，所以q OR r的值为TRUE。

现在来处理带有AND运算符的节点。它的两个子节点分别对应表达式p和q OR r，因此都具有值TRUE。所以表示表达式p AND (q OR r)的该节点的值也是TRUE。

最后，我们要继续处理带有运算符OR的根节点。我们已经得出它的左子节点的值为TRUE，而它的右子节点表示表达式s根据真值赋值其值为FALSE。因为TRUE OR FALSE求出的值是TRUE，所以整个表达式的值为TRUE。

12.3.3　布尔函数

任何表达式的“含义”都可以描述为从其参数的值到整个表达式的值的函数。例如，算术表达式$x\times(x+y)$是接受x和y（假如x和y是实数）的值，然后返回将两个参数相加并将和乘以第一个参数所得到的值。这一行为就类似如下C语言函数的行为

```
float foo(float x, float y)
{
    return x*(x+y);
}
```

在第7章中我们了解到，函数是定义域和值域有序对的集合。我们还可以把$x\times(x+y)$这样的算术表达式表示为定义域为实数对且值域为实数的函数。该函数是由形如$\big((x,y),x\times(x+y)\big)$的有序对构成的。请注意，各有序对的第一个组分本身也是有序对(x,y)。该集合是无限集，它所含的成员类似$\big((3,4),21\big)$或$\big((10,12.5),225\big)$。

类似地，逻辑表达式的含义就是接受真值赋值作为参数，并返回TRUE或FALSE的函数。这样的函数就叫作布尔函数。例如，逻辑表达式

E：p AND (p OR q)

就类似如下C语言函数

```
BOOLEAN foo(BOOLEAN p, BOOLEAN q)
{
    return p && (p || q);
}
```

和算术表达式一样，布尔表达式也可以视作有序对的集合。各有序对的第一个组分是一种

真值赋值，也就是以某指定次序为各命题变量给出真值的元组。而该有序对的第二个组分是表达式对应此真值赋值时的值。

✦ 示例 12.4

表达式$E = p$ AND (p OR q)可以用由4个成员组成的函数表示。我们在表示真值时会把对应p的值放在对应q的值之前。那么((TRUE,FALSE),TRUE)就是将E表示为函数的集合中的一个有序对。它的含义是，当p为真且q为假时，p AND (p OR q)为真。我们可以通过示例12.3中所示的过程处理表示E的表达式树来确定该值。读者可以使用其他3种真值赋值为E求值，以此构建起E所表示的整个布尔函数。

12.3.4　习题

(1) 针对所有可能的真值赋值，为以下表达式求值，从而将它们的布尔函数表示成集合论函数。

(a) p AND (p OR q)

(b) NOT p OR q

(c) (p AND q) OR (NOT p AND NOT q)

(2) 编写C语言函数实现习题(1)中的逻辑表达式。

12.4　真值表

将布尔函数表示为*真值表*是很方便的，真值表中的各行对应着各参数真值所有可能的组合。表中有着对应各参数的列以及对应函数值的列。

p	q	p AND q
0	0	0
0	1	0
1	0	0
1	1	1

p	q	p OR q
0	0	0
0	1	1
1	0	1
1	1	1

p	NOT p
0	1
1	0

图12-2　对应AND、OR和NOT的真值表

✦ 示例 12.5

对应AND、OR和NOT的真值表如图12.2所示。这里用到了简略表示法，用1代表TRUE，用0代表FALSE，本章其他部分也将经常这样表示。因此对应AND的真值表就表示，当且仅当两个操作数都为TRUE时，结果才是TRUE，而第二个真值表则表示，当操作数有一个为TRUE或两个都为TRUE时，应用OR运算符的结果为TRUE，而第三个真值表说明，当且仅当操作数的值为FALSE时，应用NOT运算符的结果为TRUE。

12.4.1　真值表的大小

假设某布尔函数具有k个参数，那么该函数的真值赋值就是具有k个元素的表，各元素要么为TRUE，要么为FALSE。计算对应k个变量的真值赋值数就是4.2节中考虑过的为分配计数问题的例子。也就是说，我们可以为这k个项每个项分配两个真值之一。这就和用两种可选颜色粉刷k所房屋的问题是类似的，因此真值赋值的数目是2^k。

因此含k个参数的布尔函数对应的真值表有2^k行，每一行都对应一种真值赋值。例如，如果$k=2$，则真值表有4行，分别对应00、01、10和11，正如我们在图12-2中看到的对应AND和OR的真值表那样。

尽管涉及两三个变量的真值表相当小。但k元函数对应2^k行这一事实说明，不用等到k变得特别大，绘制真值表就会很难了。例如，含10个参数的函数就有逾1000行。在后面几节中我们还将了解到，尽管真值表是有限的，而且原则上讲可以表示出我们想知道的与布尔函数有关的一切，但它们呈指数式增长的大小通常迫使我们寻找其他理解、比较布尔函数或为其求值的方法。

理解“蕴涵”

蕴涵（implication）运算符→的含义可能不那么直观，因为必须利用到“假蕴涵一切”的概念。我们不应该把→和因果关系混为一谈。也就是说，$p \to q$可能为真，但p并不是在任何情况下都会“导致”q。例如，设p是“天在下雨”，q是“苏带着伞”。我们可以断言$p \to q$为真。而且看起来似乎就是下雨导致苏带上她的伞。不过，也有可能苏是那种不相信天气预报而且不会一直带上雨伞出门的人。

12.4.2　布尔函数数量的计算

含k个参数的布尔函数对应真值表的行数是以k呈指数增长的，而不同k元布尔函数的数量增长得更快。要计算k元布尔函数的数量，可以注意到，正如我们所见，每个这样的函数都是由具有2^k行的真值表表示的。每一行都会被赋予一个值，要么为TRUE，要么是FALSE。因此，含k个参数的布尔函数的数量就与具有2个值的2^k项的分配的数量相同。这一数字是2^{2^k}。例如，当$k=2$时，就有$2^{2^2}=16$个函数，而对$k=5$，存在$2^{2^5}=2^{32}$，或者说是大约40亿个函数。

在含两个参数的16种布尔函数中，我们已经遇到过其中的两个：AND和OR。其他一些函数中有些是微不足道的，比如不管参数为什么值都为1的函数。不过，还有一些双参数函数是很实用的，而且我们将在本节之后的内容中看到它们。我们还看到了实用的单参数函数NOT，而且大家也经常会用到具有3个或更多参数的布尔函数。

12.4.3　更多逻辑运算符

还有以下4种双参数的布尔函数是非常实用的。

(1) 蕴涵（implication），写为→。$p \to q$的含义是，“如果p为真，那么q为真。”对应→的真值表如图12-3所示。请注意，只有在p一定为真而且q一定为假的情况下，$p \to q$才可以为假。如果p为假，那么$p \to q$恒为真，而且如果q为真，则$p \to q$恒为真。

p	q	$p \to q$
0	0	1
0	1	1
1	0	0
1	1	1

图12-3　对应“蕴涵”的真值表

(2) 等价（equivalence），写为≡，意思是“当且仅当”，即只有当p和q都为真或都为假时，才有$p \equiv q$。它的真值表如图12-4所示。另一种看待≡运算符的方式是，它表明左边和右边的操作数具有相同的真值。这就是12.2节中我们在声称 p OR((NOT p) AND q) ≡ (p OR q) 时所要表达的意思。

(3) NAND运算符，或者说“与非”运算符，是首先对操作数应用AND，然后对得到的结果应用NOT运算符求补。p NAND q就表示NOT (p AND q)。

(4) 类似地，NOR运算符，或者说“或非”运算符，是先对操作数取OR，然后对得到的结果求补，p NOR q就表示NOT (p OR q)。NAND和NOR的真值表也如图12-4所示。

p	q	$p \equiv q$
0	0	1
0	1	0
1	0	0
1	1	1

p	q	p NAND q
0	0	1
0	1	1
1	0	1
1	1	0

p	q	p NOR q
0	0	1
0	1	0
1	0	0
1	1	0

图12-4 对应等价、NAND和NOR的真值表

12.4.4 具有多个参数的运算符

一些逻辑运算符可以自然地扩展为接受两个以上参数。例如，不难看出AND是有结合性的，也就是(p AND q) AND r等价于p AND (q AND r)。因此，形如p_1 AND p_2 AND⋯AND p_k的表达式能以任意次序组合，只有在p_1、p_2、⋯、p_k都为真TRUE时，它的值才为TRUE。因此我们可以把该表达式写为具有k个参数的函数

AND($p_1, p_2, \cdots, p_k$)

它的真值表如图12-5所示。正如我们所见，只有在所有参数都是1时，结果才是1。

p_1	p_2	$\cdots$	p_{k-1}	p_k	AND $(p_1, p_2, \ldots, p_k)$
0	0	$\cdots$	0	0	0
0	0	$\cdots$	0	1	0
0	0	$\cdots$	1	0	0
0	0	$\cdots$	1	1	0
.	.		.	.	.
.	.		.	.	.
.	.		.	.	.
1	1	$\cdots$	1	0	0
1	1	$\cdots$	1	1	1

图12-5 对应k参数AND的真值表

一些运算符的重要性

我们对k元运算符AND、OR、NAND和NOR特别感兴趣的原因在于，这些运算符是特别容易以电子形式实现的。也就是说，它们是构建“门”（接受k个输入并产生这些输入的AND、OR、NAND和NOR的电子电路）的简单方式。尽管底层电子技术的细节不在本书要介绍的范围之内，但其思路通俗来说，就是用两种不同的电压表示1和0（即TRUE和FALSE）。其他一些运算符，比如≡和

→，就不是很容易用电子方式实现，而我们一般会使用若干个NAND或NOR门来实现它们。不过，NOT运算符既可看作单参数的NAND，也可看作单参数的NOR，因此也是“很容易”实现的。

同样，OR也是具有结合性的，我们可以把逻辑表达式p_1 OR p_2 OR ⋯ OR p_k表示成布尔函数OR ($p_1,p_2,\cdots p_k$)。对应k元OR的真值表有 2^k 行，就像k元AND的真值表那样。不过，对这一k元OR的真值表来说，只有p_1、p_2、⋯、p_k都被赋值为0的第一行的值才是0，而其余 2^k-1行的值全为1。

二元运算符NAND和NOR是可交换但不可结合的。因此p_1 NAND p_2 NAND⋯NAND p_k这个不含括号的表达式是没有固有含义的。在讲到k元NAND时，并不表示

p_1 NAND p_2 NAND⋯NAND p_k

任何可能的分组。而是把NAND ($p_1,p_2,\cdots,p_k$)定义为

NOT (p_1 AND p_2 AND⋯AND p_k)

也就是说，只有在p_1、p_2、⋯、p_k的值都为1时，NAND ($p_1,p_2,\cdots,p_k$)的值才是0，对其他 2^K-1 种输入组合而言，其值都为1。

同样，NOR($p_1,p_2,\cdots,p_k$)表示NOT (p_1 OR p_2 OR⋯OR p_k)。只有在p_1、p_2、⋯、p_k的值全部是0时，它的值才是1，否则它的值为0。

12.4.5　逻辑运算符的结合性与优先级

我们将用到的优先级次序是

(1) NOT（最高）

(2) NAND

(3) NOR

(4) AND

(5) OR

(6) →

(7) ≡（最低）

因此，举例来说，$p \rightarrow p \equiv$ NOT p OR q被分组为$(p \rightarrow p) \equiv ((\text{NOT } p) \text{OR } q)$。

正如我们之前提过的，AND和OR，以及≡，都是具有结合性和交换性的。如果有必要指定的话，一般会假设它们是从左起组合的。我们一般会明确地给出括号，以防出现歧义，不过→、NAND和NOR这样的运算符在两个或多个相同运算符组成的串中都是从左起组合的。

12.4.6　利用真值表为逻辑表达式求值

只要表达式E中不含太多变量，利用真值表针对所有可能的真值赋值计算和展示E的值就是一种方便的方法。我们首先有对应E中各变量的列，然后是按照从下到上为E的表达式树求值的次序对应E各子表达式的列。

在对表示某些节点的值的列应用运算符时，我们会用一种简单的方式为对应该运算符的列执行运算。例如，如果希望对两列取AND，就在两列都为1的那行中放上1，并在其他行中放上0。要是为两列求OR，就要在其中一列或者两列都为1的那几行中放上1，并在其他行中放上0。如果要为一列取NOT，就是为该列求补，如果那列有0，则放上1，反之亦然。最后再举个例子，对两列应用→运算符，只有在第一列为1且第二列为0时，其结果才是0，结果中的其他各行都是1。

其他一些运算符的规则留作本节习题。一般而言，我们会通过一行行地对所在行中各值应用运算符，以对各列应用运算符。

✦ 示例 12.6

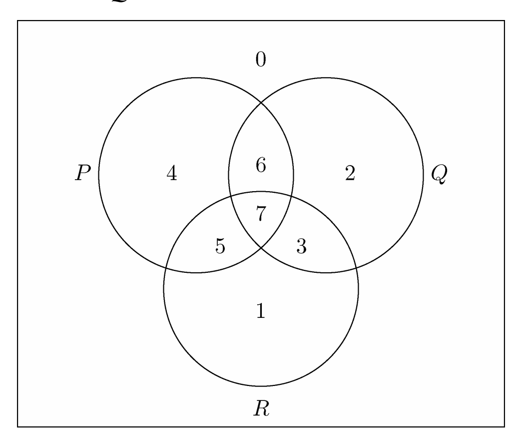
考虑表达式E：(p AND q) → (p OR r)。图12-6给出了对应该表达式及其子表达式的真值表。(1)、(2)、(3)这3列给出了变量p、q和r的值的所有组合。第(4)列给出了子表达式p AND q的值，只要第(1)和第(2)列的值都是1，该列的值就是1。而第(5)列给出了子表达式p OR r的值，在第(1)列或第(3)列为1，或者第(1)和第(3)列都为1时，第(5)列的值是1。最后，第(6)列表示整个表达式E的值。它是通过第(4)和第(5)列得到的，除了第(4)列为1且第(5)列为0的情况外，这列的值都是1。因为不存在这样的行，所以第(6)行全是1，也就是说不管参数是什么，E的真值都是1。正如我们将在12.7节中看到的，这样的表达式称为"重言式"。

(1)	(2)	(3)	(4)	(5)	(6)
p	q	r	p AND q	p OR r	E
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	0	0	1
0	1	1	0	1	1
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	1	1	1
1	1	1	1	1	1

图12-6　对应(p AND q) → (p OR r)的真值表

文氏图和真值表

真值表与7.3节中讨论过的表示集合运算的文氏图之间存在着相似性。首先，并集运算就类似于真值的OR，而交集则类似AND。我们将在12.8节中看到，这两对运算满足相同的代数法则。就像涉及k个集合作为参数的表达式会把文氏图分成2^k个区域那样，具有k个变量的逻辑表达式也会形成具有2^k行的真值表。此外，在这些区域与这些真值表行之间也存在自然的对应。例如，具有变量p、q和r的逻辑表达式就对应涉及P、Q和R这3个集合的集合表达式。考虑对应这些集合的文氏图：



在这里，区域0对应不在P、Q、R任意一个中的元素构成的集合。区域1则对应着在R中但不在P或Q中的元素。一般地讲，如果考虑3位区域编号的二进制表示，比方说是abc，那么如果$a=1$，则表示元素在P中，如果$b=1$则在Q中，而如果$c=1$则在R中。因此，编号为$(abc)_2$的

区域就对应着p、q、r分别具有真值a、b、c时真值表的那行。

在处理文氏图时，表示两个集合并集的区域会包含对应两者中任一集合的区域。与此相似的是，在为真值表中的列求OR时，我们会在第一列有1的行与第二列有1的行的并集中放上1。类似地，文氏图中表示集合交集的区域就是只取重在这两个集合中的区域，而为列求AND就是在第一列有1的行与第二列有1的行的交集中放上1。

逻辑运算符NOT与集合运算符没有太多对应。不过，如果将所有区域的并集想象成个“全集”，那么逻辑NOT对应着取走一些区域，并生成文氏图中剩余区域组成的集合，也就是从全集中减去给定的集合。

12.4.7 习题

(1) 给出计算真值表中两列的(a) NAND；(b) NOR；(c) ≡的规则。

(2) 为以下表达式及它们的子表达式计算真值表。

(a) $(p \to q) \equiv (\text{NOT}\ p\ \text{OR}\ q)$

(b) $p \to (q \to (r\ \text{OR NOT}\ p))$

(c) $(p\ \text{OR}\ q) \to (p\ \text{AND}\ q)$

(3) * 逻辑表达式p AND NOT q对应什么集合运算符？（见之前比较文氏图和真值表的短文。）

(4) * 给出说明→、NAND和NOR不具结合性的例子。

(5) ** 如果布尔函数f满足$f(\text{TRUE}, x_2, x_3, \cdots, x_k) = f(\text{FALSE}, x_2, x_3, \cdots, x_k)$，则说$f$是不依赖第一个参数的。同样，如果$f$的第$i$个参数在TRUE和FALSE之间变换却不会使$f$的值改变，就可以说$f$是不依赖第$i$个参数的。有多少双参数布尔函数是不依赖它们的第一个或第二个参数（或两个参数都不依赖）的？

(6) * 为具有两个变量的16种布尔函数构建真值表。这些函数中有多少种具有交换性？

(7) 二元异或（exclusive-or）函数 ⊕ 的定义是，当且仅当只有其中一个参数为TRUE时其值为TRUE。

(a) 画出 ⊕ 的真值表。

(b) ⊕ 是否具有交换性？它是否具有结合性？

12.5 从布尔函数到逻辑表达式

现在考虑从真值表设计逻辑表达式的问题。从作为逻辑表达式规范的真值表开始，目标则是找到具有给定真值表的表达式。一般而言，可以利用无数个不同的表达式，我们往往将选择限制到特定的运算符集合中，而且通常会希望表达式从某种意义上讲是“最简单的”。

该问题是电路设计中的基础问题。表达式中的逻辑运算符可以被理解成电路的门，这样的话就存在从逻辑表达式到电子电路的直接转化，这种转化是通过第13章将要讨论的过程实现的。

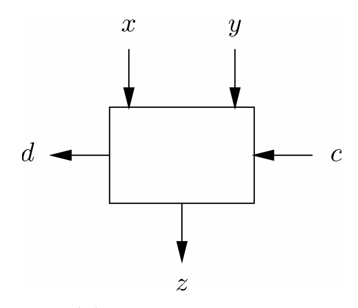


图12-7 一位加法器：$(dz)_2$ 是 $x+y+c$ 的和

✦ 示例 12.7

正如我们在1.3节中看到的，可以用图12-7所示的这种一位加法器设计32位加法器。一位加法器会把两个输入位x和y与进位输入位c相加，得到进位输出位d与和值位z。

图12-8中的真值表给出了进位输出位d与和值位z的值，将其表示为对应8种输入值组合的x、y、c的函数。如果x、y和c中至少有两个的值是1，进位输出位d就是1，而只有在输入中没有1或者只有一个1时，才有$d = 0$。如果x、y和c中有奇数个为1，和值位z就是1，否则就是0。

	x	*y*	*z*	*d*	*z*
0)	0	0	0	0	0
1)	0	0	1	0	1
2)	0	1	0	0	1
3)	0	1	1	1	0
4)	1	0	0	0	1
5)	1	0	1	1	0
6)	1	1	0	1	0
7)	1	1	1	1	1

图12-8　对应进位输出位d与和值位z的真值表

我们要展示一种从真值表立即转换成逻辑表达式的一般性方法。不过，给定图12-8中进位输出函数d的情况下，可以按照如下方式进行推理，构建对应的逻辑表达式。

(1) 从第3行和第7行可知，如果y和c都是1，则d是1。

(2) 从第5行和第7行可知，如果x和c都是1，则d是1。

(3) 从第6行和第7行可知，如果x和y都是1，则d是1。

条件(1)可以用逻辑表达式y AND c模拟，因为y AND c只有在y和c都是1时才为真。同样，条件(2)可以用x AND c模拟，而条件(3)则可通过x AND y模拟。

所有有d = 1的行都是这3种情况中的某一行。因此可以写出一个逻辑表达式，它只要在这3个条件中至少有一个成立的情况下为真即可，也就是要为这3个表达式取逻辑OR：

$$(y \text{ AND } c) \text{ OR } (x \text{ AND } c) \text{ OR } (x \text{ AND } y) \tag{12.5}$$

这一表达式的正确性在图12-9中得到了验证。后4列分别对应子表达式y AND c、x AND c、x AND y和表达式(12.5)。

x	*y*	*c*	*y* AND *c*	*x* AND *c*	*x* AND *y*	*d*
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	1	0	0	1
1	0	0	0	0	0	0
1	0	1	0	1	0	1
1	1	0	0	0	1	1
1	1	1	1	1	1	1

图12-9　对应进位输出表达式(12.5)及其子表达式的真值表

12.5.1　简化符号

在继续描述如何从真值表构建表达式之前，我们要对表示法进行一些有意义的简化。

(1) 可以通过直接并列（也就是不使用任何运算符）表示AND运算符，就像表示乘法，以及第10章中表示串接时那样。

(2) OR运算符可以表示为+。

(3) NOT运算符可以用上横线表示。这种约定在NOT应用到单个变量上时特别实用，我们经常把NOT p写为 $\bar{p}$ 。

✦ 示例 12.8

表达式p AND q OR r可以写为pq+r，表达式p AND NOT q OR NOT r则可以写为 $p\bar{q}+\bar{r}$ 。我们甚至可以将原始符号与简化符号混用。例如，表达式

$$((p \text{ AND } q) \rightarrow r) \text{ AND } (p \rightarrow s)$$

可以写成$(pq \rightarrow r)$ AND $(p \rightarrow s)$，甚至可以写成$(pq \rightarrow r)\ (p \rightarrow s)$。

使用这种新表示法的一个重要原因在于，这样可以让我们把AND和OR视作算术运算中的乘法和加法。因此可以应用诸如交换律、结合律和分配律这样的类似法则，在12.8节中我们将会看到这些法则适用于这些逻辑运算符，就像这些法则对相应的算术运算符所做的那样。例如，我们会看到$p(q+r)$可以被$pq+pr$替换，然后被$rp+qp$替换，不管涉及的运算符是AND和OR，还是乘法和加法。

因为有了这种简化符号，通常可以把表达式的AND称为*积*，把表达式的OR称为*和*。表达式的AND也可以称为*合取*（conjunction），而表达式的OR还可以叫作*析取*（disjunction）。

12.5.2　从真值表构建逻辑表达式

任何布尔函数都可以用使用AND、OR和NOT运算符的逻辑表达式表示。为给定的布尔函数找到最简单的表达式一般是很难的。不过，为布尔函数构建某一表达式却很容易，用到的技巧也很简单。首先从函数的真值表开始，构建形如

$$m_1 \text{ OR } m_2 \text{ OR} \cdots \text{OR } m_n$$

的逻辑表达式。各个m_i都是与真值表中让函数的值为1的某一行对应的。因此该表达式中项数与表示函数的那列中1的个数是相等的。这些m_i项都被叫作*最小项*（minterm），并具有下面将要描述的特殊形式。

要开始对最小项的解释，首先要提到*文字*（literal），这里的文字要么为单个命题变量（比如p），要么为求反变量（比如我们一般会写为 $\bar{p}$ 的NOT p）。如果真值表中有k列表示变量的列，那么每个最小项都是由k个文字的逻辑AND（或“积”）表示的。设r是我们想为其构建最小项的某一行。如果变量p在行r的值是1，就选择文字p。如果p在行r的值是0，则选择 $\bar{p}$ 作为文字。行r的最小项就是各变量对应文字的积。明确地讲，如果所有变量都有真值表行r中的值，那么最小项的值就只可能是1。

现在要通过为与函数值为1的行对应的最小项求逻辑OR（或“和”），来为函数构建表达式。得到的表达式具有“积的和”的形式，或者说它是*析取范式*（disjunctive normal form）。该表达式是正确的，因为只有在存在值为1的最小项时，它的值才是1，而除非变量的值对应着真值表中该最小项所在的那行，而且该行的值为1，否则该最小项不可能为1。

✦ 示例 12.9

我们来为由图12-8中的真值表所定义的进位输出函数d构建析取范式。值为1的行的编号分别是3、5、6和7。第3行有$x=1$、$y=1$和$c=1$，因此该行的最小项是 $\bar{x}$ AND y AND c，可以将其简

写为 $\bar{x}yc$ 。类似地，第5行的最小项是 $x\bar{y}c$ ，第6行的最小项是 $xy\bar{c}$ ，而第7行的最小项是xyc。因此所需的对应d的表达式就是这些表达式的逻辑OR，也就是

$$\bar{x}yc + x\bar{y}c + xy\bar{c} + xyc \tag{12.6}$$

这一表达式要比(12.5)更复杂。不过，我们将在12.6节中看到如何得出表达式(12.5)。

同样，通过把对应第1、2、4和7行的最小项相加，可以为和值位z构建逻辑表达式，得到

$$\bar{x}\,\bar{y}c + \bar{x}y\bar{c} + x\bar{y}\,\bar{c} + xyc$$

运算符的完全集

用来设计(12.6)式这样析取范式的最小项技术表明，逻辑运算符AND、OR和NOT的集合是完全集，就是说，每个布尔函数都具有只使用这3种运算符的表达式。不难证明NAND本身也是完全的。我们可以将涉及AND、OR和NOT的函数只用NAND表示成如下这样。

(1) $(p \text{ AND } q) \equiv \big((p \text{ NAND } q) \text{ NAND TRUE}\big)$

(2) $(p \text{ OR } q) \equiv \big((p \text{ NAND TRUE }) \text{ NAND } (q \text{ NAND TRUE })\big)$

(3) $(\text{ NOT } p) \equiv \big(p \text{ NAND TRUE}\big)$

通过用合适的NAND表达式来替换用到AND、OR和NOT的地方，可以把任何析取范式转换成只涉及NAND的表达式。同样，NOR自身也是完全的。

由运算符AND和OR构成的集合就不是完全集。比方说，它们没法表示函数NOT。要知道原因，我们可以注意到AND和OR都是单调的，这就是说，在把任何一个输入从0变为1时，输出都不能从1变成0。可以通过对表达式的大小进行归纳，证明任何只有AND和OR运算符的表达式都是单调的。不过NOT显然不是单调的，因此没办法只用AND和OR表示NOT。

---

p	q	r	a	b
0	0	0	0	1
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

图12-10 用于习题的两个布尔函数

12.5.3 习题

(1) 图12-10是用变量p、q和r定义a和b这两个布尔函数的真值表，为这两个函数分别写出析取范式。

(2) 为下列函数写出合取范式（见下面的附注栏“和的积表达式”）。

(a) 图12-10中的函数a。

(b) 图12-10中的函数b。

(c) 图12-8中的函数z。

和的积表达式

有两种方式可以把真值表转换成涉及AND、OR和NOT的表达式，这里的表达式将会是文字之和（逻辑OR）的积（逻辑AND）。这种形式就叫作“和的积”，或合取范式（conjunction normal form）。

对真值表的各行而言，我们可以定义最大项，它是与所在行中某一参数变量的值不相同的文字的和。也就是说，如果该行中变量p的值是0，就使用文字p，如果那行中p的值为1，就使用$\bar{p}$。因此，除非每个变量p都具有该行指定给p的值，否则最大项的值就是1。

因此，如果查看真值表中值为0的各行，并为这些行的最大项取逻辑AND，该表达式就只会在输入匹配函数值为0的某一行时值为0。这样一来，该表达式对其他各行而言值都为1，也就是对真值表中函数值为1的那些行来说都是1。例如，图12-8的真值表中，第0、1、2和4行对应d的值为0。比方说，第0行的最大项就是$x+y+c$，而第1行的最大项就是$x+y+\bar{c}$，所以d的合取范式就是

$$(x+y+c)(x+y+\bar{c})(x+\bar{y}+c)(\bar{x}+y+c)$$

该表达式与(12.5)和(12.6)式都是等价的。

(3) ** 以下哪个逻辑运算符可以单独形成运算符完全集：$(a)\equiv$;$(b)\rightarrow$;(c) NOR？在每种情况中都对自己的答案加以证明。

(4) ** 在16个双变量的布尔函数中，有多少函数自身就是完全的？

(5) * 证明，单调函数的AND和OR还是单调的。然后证明只含AND和OR运算符的表达式都是单调的。

12.6　利用卡诺图设计逻辑表达式

在本节中，我们要展示一种为布尔函数确定析取范式的制表技巧。用这种方法生成的表达式通常要比12.5节中通过为真值表中所有必要的最小项求逻辑OR这样的权宜之计所构建出的表达式更简单。

举例来说，在示例12.7中，我们为一位加法器的进位输出函数对应的表达式进行了专门设计。可以看到，有可能使用不是最小项的文字之积，也就是说，缺少与某些变量对应的文字。例如，可以用文字之积xy来涵盖图12-8中的第(6)和第(7)两行，因为只有在变量x、y和c具有这两行中的某一行所表示的值时，xy的值才是1。

同样，在示例12.7中，我们使用了表达式xc来涵盖第(5)和第(7)行，并用yc涵盖第(3)和第(7)行。请注意，所有3个表达式都涵盖了第7行。不过这并没有什么坏处。其实，假如分别只使用对应第(5)和第(3)行的最小项，也就是$x\bar{y}c$和$\bar{x}yc$，来替代xc和yc，我们会得到正确的表达式，但它就要比示例12.7中得到的表达式$xy+xc+yc$多两个运算符。

这里的基本概念就是，如果两个最小项唯一的区别是某一个变量的值相反，比如第(6)和第(7)行的$xy\bar{c}$和xyc，就可以通过取相同的文字并去掉那个有区别的变量，把两个最小项结合起来。这一结论遵从以下一般法则

$$(pq+\bar{p}q)\equiv q$$

要理解这一等价性，就要注意到如果q为真，那么要么pq为真。要么$\bar{p}q$为真。而且反过来，如果pq或$\bar{p}q$有一个为真，那么一定有q为真。

12.7节中介绍了验证这些法则的技巧，不过现在只要其直觉含义支撑其使用即可。还要注意到，该法则的使用并不仅限于最小项。例如，可以设p是任意命题变量，而q是任意的文字之

积。你可以合并任何两个只有一个变量不同的文字积（一个积含有变量本身，另一个则含有其互补变量），用由相同文字组成的一个积代替这两个积。

12.6.1 卡诺图

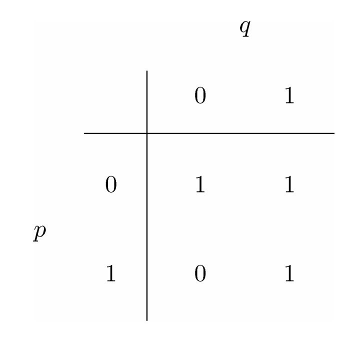
有一种制图技巧，可以根据真值表设计析取范式，这种方法对最多含4个变量的布尔函数来说效果甚佳。这种思路就是把真值表写成名为卡诺图（Karnaugh map）的二维数组，该二维数组的项（或者说“点”）各自表示真值表中的行。通过让只有一个变量不同的行所对应的点保持邻接，可以把有用的文字积看作某些矩形，而这些矩形中的点的值都是1。

12.6.2 双变量卡诺图

最简单的卡诺图是对应双变量布尔函数的。各行对应着其中一个变量的值，而各列对应另一个变量的值。图中的项是0或1，取决于两个变量值的组合使函数的值为0还是为1。因此，该卡诺图是双变量布尔函数真值表的二维表示。

✦ 示例 12.10

在图12-11中，我们看到表示“蕴涵”函数 $p \to q$ 的卡诺图。其中4个点分别对应着p和q的值4种可能的组合。请注意，除了 $p=1$ 且 $q=0$ 的情况之外，“蕴涵”的值都是1，因此，卡诺图中值为0的点只有对应 $p=1$ 且 $q=0$ 的那项，其他点的值都是1。



p \ q	0	1
0	1	1
1	0	1

图12-11 表示 $p \to q$ 的卡诺图

12.6.3 蕴涵项

布尔函数f的蕴涵项（implicant）是一些文字的积x，它满足的条件是：f中变量的任何赋值组合都不能使x为真且f为假。例如，每一个让函数f的值是1的最小项都是f的蕴涵项。不过，其他积也可以是蕴涵项，我们将会了解如何从f的卡诺图中解读这些蕴涵项。

✦ 示例 12.11

最小项pq是图12-11中“蕴涵”函数的蕴涵项，因为让pq为真的变量赋值组合（即 $p=1$ 且 $q=0$ ）也能使“蕴涵”函数为真。

再举个例子，$\bar{p}$ 本身也是“蕴涵”函数的蕴涵项，因为使为真的两种p和q赋值组合，也能让 $p \to q$ 为真。这两种赋值组合分别是 $p=0$ 且 $q=0$ ，以及 $p=0$ 且 $q=1$ 。

蕴涵项涵盖了函数值为1的那些点。通过为涵盖了所有令函数值为1的点的蕴涵项取`OR`，就

可以为布尔函数构建逻辑表达式。

✦ 示例 12.12

图12-12展示了对应"蕴涵"函数的卡诺图中的两个蕴涵项。较大的那个涵盖了两个点，对应着单个文字 $\bar{p}$ 。这一蕴涵项涵盖了卡诺图中顶部的两个点，这两个点的值都是1。而较小的蕴涵项pq涵盖了 $p=1$ 且 $q=1$ 的那个点。因为这两个蕴涵项加在一起涵盖了所有值为1的点，所以它们的和 $\bar{p}+pq$ 就是与 $p \to q$ 等价的表达式，也就是说 $(p \to q) \equiv (\bar{p}+pq)$ 。

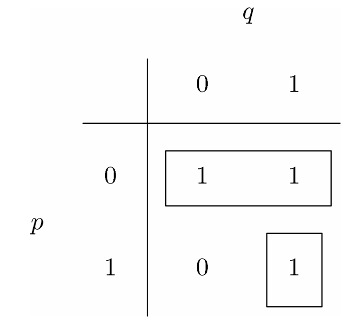


图12-12　表示 $p \to q$ 的卡诺图中的两个蕴涵项 $\bar{p}$ 和pq

与卡诺图中的蕴涵项对应的矩形必须具有特殊的"外观"。对源自双变量函数的简单卡诺图来说，这些矩形只可能是下列之一。

(1) 单个点；

(2) 某行或某列；

(3) 整个图。

卡诺图中的单个点对应着最小项，通过为该点所在行和列相应变量对应的文字求积，便可以得出其表达式。也就是说，如果该点所在的行或列是0，就可以分别为该行或该列对应的变量取反。如果该点在对应1 的行或列中，就取对应的变量本身。例如，图12-12中较小的蕴涵项就在 $p=1$ 的那行和 $q=1$ 的那列中。这就是我们要用非否定文字p和q的积作为该蕴涵项的原因。

双变量卡诺图中行或列对应着两个对一个变量相同而对另一个变量相反的点。与之对应文字的"积"就减少为单个文字。剩下的这个文字具有共同值为这些点所共享的变量。如果该共同值是0，那么该文字就是否定的，而如果共享的值是1，该文字就是非否定的。因此，图12-12中较大的蕴涵项，即第一行，其中的点具有相同的p值。该值是0，这样就说明为该蕴涵项使用文字积 $\bar{p}$ 是合理的。

由整个图组成的蕴涵项是种特例。原则上讲，这对应着积退化为常数1，或者说`TRUE`的情况。显然，对应逻辑表达式`TRUE`的卡诺图在图中所有点的位置都是1。

12.6.4　质蕴涵项

如果布尔函数f的蕴涵项x在删除其中任何文字后不再为蕴涵项，则x就是f的*质蕴涵项*（prime implicant）。事实上，质蕴涵项就是所含文字尽可能少的蕴涵项。

请注意，这样的矩形越多，其积中文字的数量就越少。我们一般会选择用文字较少的积替换具有很多文字的积，文字较少的积涉及的运算符更少，因此"更加简单"。所以我们在选择若干蕴涵项涵盖卡诺图时最好只考虑那些质蕴涵项。

请记住，对应某给定卡诺图的每个蕴涵项都只由值为1的点组成。一个蕴涵项之所以是质蕴涵项，是因为使其大小翻番可能会迫使我们融入一个值为0的点。

✦ 示例 12.13

在图12-12中，较大的蕴涵项 $\bar{p}$ 是质蕴涵项，因为唯一可能比它还大的蕴涵项是全图，而后者不可能是蕴涵项，因为全图中含有0。较小的蕴涵项pq不是质蕴涵项，因为它被包含在只由1组成、同为该“蕴涵”卡诺图蕴涵项的第二列中。图12-13展示出了该“蕴涵”图仅有的质蕴涵项。[①]它们对应着积 $\bar{p}$ 和q，而且它们可以进一步组成表达式 $\bar{p}+q$ ，我们在12.3节中就注意到这一表达式是与 $p \to q$ 等价的。

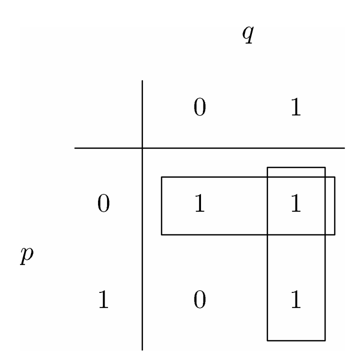


图12-13 对应“蕴涵”函数的质蕴涵项 $\bar{p}$ 和q

12.6.5 三变量卡诺图

当真值表中有3个变量时，我们可以使用图12-14这样两行四列的图，该图对应图12-8所示的进位输出真值表。请注意，与两个变量（本例中是变量y和c）的值对对应的各列是按照一种特别的次序排列的。原因在于，我们希望邻接的列对应的真值赋值只有一个变量是不同的。假如按照一般的顺序00、01、10、11来排列，中间的两列就会有y和c两个变量是不同的。还要注意，第一列和最后一列也是“邻接的”，这样它们只有变量y是不同的。因此，当我们选择蕴涵项时，可以把第一列和最后一列看作 2×2 的矩阵，而且可以把每行的第一个点和最后一个点当作 1×2 的矩阵。

x \ yc	00	01	11	10
0	0	0	1	0
1	0	1	1	1

图12-14 对应进位输出函数的卡诺图，其中质蕴涵项是xc、yc和xy

① 一般来讲，可能有多个可以涵盖某给定卡诺图的质蕴涵项集合。

我们需要推导该三变量卡诺图有哪些矩形表示可能的蕴涵项。首先，许可的矩形必须对应文字的积。在任何积中，各变量只可能以如下3种方式之一出现：否定的、非否定的，或根本没有。当变量是否定的或非否定的时，它会让对应蕴涵项的点数减半，因为只有具有该变量合适的值的点才属于该蕴涵项。因此，蕴涵项中的点数总是2的乘方。因此，对各变量而言，许可的蕴涵项是满足以下条件之一的若干个点。

(1) 只包含该变量等于0的点；

(2) 只包含该变量等于1的点；

(3) 该变量是什么值都没有区别。

从卡诺图中解读蕴涵项

不管涉及多少个变量，都可以取任一表示蕴涵项的矩形，并生成只对该矩形中的点来说为TRUE的文字积。如果p是任意变量，那么

(1) 如果该矩形中的每个点都有 $p=1$，那么p是该积中的文字。

(2) 如果该矩形中的每个点都有 $p=0$，那么 $\overline{p}$ 是该积中的文字。

(3) 如果该矩形中某些点有 $p=0$，而另一些点有 $p=1$，那么该积中没有变量p的文字。

对三变量卡诺图而言，可以按照以下方式列举可能的蕴涵项。

(1) 任何点。

(2) 任何列。

(3) 任何一对水平邻接的点，包含末端环回的情况，也就是各行的第1列和第4列构成的一对。

(4) 任何行。

(5) 任何由两列邻接列组成的 2×2 正方形，包括末端环回的情况，也就是第1和第4列。

(6)整个图。

✦ 示例 12.14

对应进位输出函数的3个质蕴涵项如图12-14所示。我们可以将各质蕴涵项转换成文字之积，详细方法见上文附注栏内容“从卡诺图中解读蕴涵项”。对应的积是最左边的xc，垂直的yc和最右边的xy。这3个表达式的和就是我们在示例12.7中用非正式方法得出的析取范式，你现在应该就明白这一表达式是怎么得出的了。

✦ 示例 12.15

图12-15展示了与三变量布尔函数NAND(p, q, r)对应的卡诺图。质蕴涵项有

(1) 第一行，对应 $\overline{p}$ ；

(2) 前两行，对应 $\overline{q}$ ；

(3) 第1和第4列，对应 $\overline{r}$ 。

因此该卡诺图的析取范式是 $\overline{p}+\overline{q}+\overline{r}$ 。

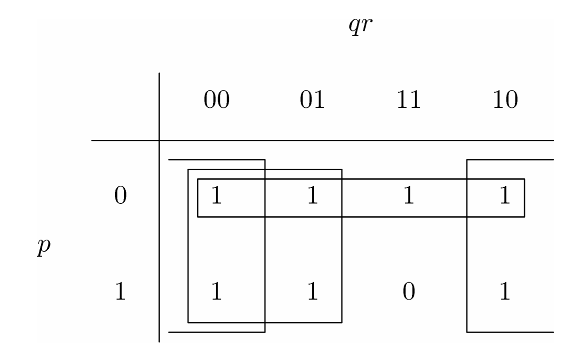


图12-15 NAND(p, q, r)的卡诺图，其中质蕴涵项是 $\overline{p}$ 、$\overline{q}$ 和 $\overline{r}$

12.6.6 四变量卡诺图

四参数函数可以用4×4 的卡诺图表示，该图中两个变量对应各行，另两个变量对应各列。对图中的行和列，我们必须用到之前为三变量卡诺图的列排序时所用到的次序，得到的四变量卡诺图就如图12-16所示。对四变量卡诺图来说，行和列的邻接都要考虑到末端环回的情况。也就是说，顶部的行和底部的行是邻接的，最左的列和最右的列是邻接的。作为一个重要的特例，4个角上的点也形成了一个 2×2 的矩形，它们对应着图12-16中的文字之积 $\overline{q}\,\overline{s}$ （这不是图12-16中的蕴涵项，因为右下角是0）。

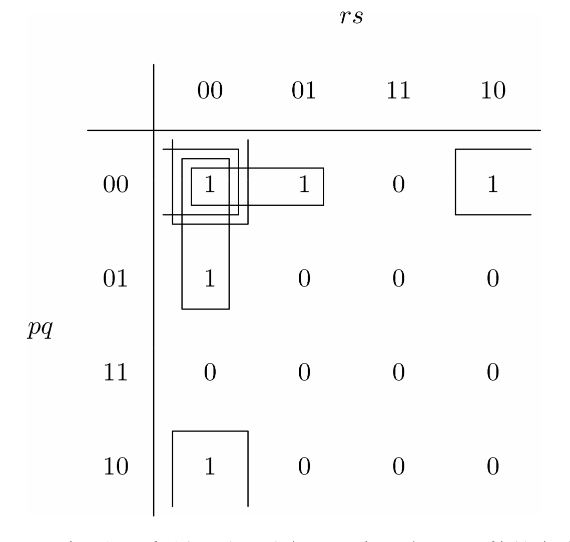


图12-16 标示了质蕴涵项，对应“至多一个1”函数的卡诺图

四变量卡诺图中对应文字积的矩形分别为：

(1) 任何点；

(2) 任何两个水平或垂直方向上的邻接点，包含那些末端环回情况下的邻接点；

(3) 任何行或列；

(4) 任何 2×2 正方形，包括那些末端环回的情况，比如顶部的那行中的两个点，以及同两列中底部那行的两个点。正如之前提过的，图中4个角上点也是这种“正方形”的一个特例；

(5) 任何 2×4 或 4×2 的矩形，包括那些末端环回的情况，比如第一列加上最后一列；

(6) 整个图。

✦ 示例 12.16

图12-16展示了具有p、q、r、s4个变量布尔函数对应的卡诺图，该函数在输入中至多有一个1时值才是1。其中有4个质蕴涵项，都是大小为2的，而且其中有两个是末端环回的。顶部那行的第一个点和最后一个点组成的蕴涵项中，两个点的p、q和s变量具有相同的值，而且各变量的共同值都是0。因此它的文字积就是 $\overline{p}\overline{q}\overline{s}$ 。而类似地，其他蕴涵项的积分别是 $\overline{p}\overline{q}\overline{r}$ 、 $\overline{p}\overline{r}\overline{s}$和 $\overline{q}\overline{r}\overline{s}$ 。所以对应该函数的表达式为

$$\overline{p}\overline{q}\overline{r}+\overline{p}\overline{q}\overline{s}+\overline{p}\overline{r}\overline{s}+\overline{q}\overline{r}\overline{s}$$

pq \ rs	00	01	11	10
00	1	1	0	1
01	0	0	0	1
11	1	0	0	0
10	1	0	1	1

图12-17　4个角组成的蕴涵项为质蕴涵项的卡诺图

✦ 示例 12.17

之所以选择图12-17中的卡诺图，是因为其中的1所具有的模式，而不是出于其函数所具有的显著特征。这幅图展示一个重点。5个质蕴涵项一起涵盖了所示的全部值是1的点，其中包括由4个角构成的蕴涵项（用虚线表示），而该蕴涵项的文字积表达式是 $\overline{q}\overline{s}$ ，另外4个质蕴涵项分别具有文字积 $\overline{p}\overline{q}\overline{r}$ 、 $\overline{p}r\overline{s}$ 、 $p\overline{q}r$ 和 $p\overline{r}\overline{s}$ 。

从目前为止的例子来看，我们可能会觉得，要为该图生成逻辑表达式，应该要将全部5个蕴涵项取逻辑OR。不过，片刻思考后你就会觉得，最大的蕴涵项 $\overline{q}\overline{s}$ 是多余的，因为所有的点都已经被其他4个质蕴涵项涵盖了。此外，它也是唯一一个可以消除的质蕴涵项，因为其他4个质蕴涵项都各自含有一个只由自身涵盖的点。例如， $\overline{p}\overline{q}\overline{r}$ 就是唯一一个涵盖了第一行第二列那个点的质蕴涵项。因此下列表达式就是从图12-17所示的卡诺图得到的所需的析取范式。

$$\overline{p}\overline{q}\overline{r}+\overline{p}r\overline{s}+p\overline{q}r+p\overline{r}\overline{s}$$

12.6.7　习题

(1) 为变量p、q、r和s的以下函数画出卡诺图。

　(a) 如果p、q、r和s中有一个、两个或三个为TRUE，则该函数为TRUE，如果没有一个为TRUE或全部为TRUE，则该函数不为TRUE。

(b) 如果p、q、r和s中至多有两个为TRUE，则该函数为TRUE，如果有三个或四个为TRUE，则该函数不为TRUE。

(c) 如果p、q、r和s中有一个、三个或四个为TRUE，则该函数为TRUE，如果没有一个为TRUE或有两个为TRUE，则该函数不为TRUE。

(d) 由逻辑表达式 $pqr \to s$ 表示的函数。

(e) 如果$pqrs$所表示的二进制数字的值小于10，则该函数为TRUE。

(2) 为习题(1)中的各个卡诺图找出除了最小项之外的蕴涵项。它们中有哪些是质蕴涵项？为各函数找出涵盖卡诺图中所有1的质蕴涵项之和。是否要用到所有的质蕴涵项？

(3) 证明，布尔函数析取范式中的每个积都是该函数的蕴涵项。

(4) * 大家还可以根据卡诺图构造合取范式。首先要找到形成蕴涵项的那种矩形，不过这里矩形中的点要全部为0，而不是全部为1。这样的矩形可以称为“反蕴涵项”。我们可以为各反蕴涵项构造一个对所有除反蕴涵项所含点之外的点而言值为1的文字和。对各变量x而言，如果相应的反蕴涵项只包含 $x=0$ 的点，则该文字和中具有文字x，而如果相应的反蕴涵项中只有那些 $x=1$ 的点，它就具有文字 $\bar{x}$ 。否则，该文字和中不含有涉及x的文字。

(5) 利用习题(4)得到的答案，为习题(1)中的各函数写出相应的合取范式。要让合取范式中包含尽可能少的文字和。

(6) ** 在 4×4 的卡诺图中，有多少构成蕴涵项的(a) 1×2 (b) 2×2 (c) 1×4 (d) 2×4 矩形？假设变量分别是p、q、r和s，把它们的蕴涵项描述成文字积的形式。

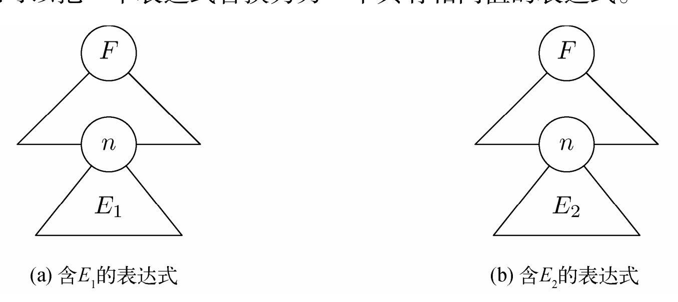
12.7 重言式

重言式（tautology）是指不管其命题变量的值如何，其值都为真的逻辑表达式。对重言式而言，真值表的所有行，或者说卡诺图中的所有点，都具有值1。简单的重言式例子包括

$$\text{TRUE}$$
$$p+\bar{p}$$
$$(p+q)\equiv(p+\bar{p}q)$$

重言式有很多重要用途。例如，假设形如 $E_1 \equiv E_2$ 这样的表达式是重言式。那么，只要在任何表达式中出现 E_1 的实例，就可以用 E_2 替换 E_1 ，而得到的表达式仍然表示相同的布尔函数。

图12-18a展示了包含子表达式 E_1 的逻辑表达式F所对应的表达式树。而图12-18b则是用 E_2 代替 E_1 的相同表达式树。原因在于，我们知道两棵树中标记为n的节点，也就是对应 E_1 和 E_2 的表达式树的根节点，在两棵树中一定有着相同的值，因为有 $E_1 \equiv E_2$ 。而为两棵树中n以上的部分求值，显然会得出相同的值，这样就证明了两棵树是等价的。这种等价表达式可以彼此替换的能力通俗点讲就是“以相等换相等”。请注意，在其他的代数，诸如算术、集合、关系或正则表达式代数中，也可以把一个表达式替换为另一个具有相同值的表达式。



(a) 含E_1的表达式　　(b) 含E_2的表达式

图12-18　展示以相等换相等的表达式树

✦ 示例 12.18

考虑逻辑运算符OR的结合律，可以将其表示为表达式

$$\big((p+q)+r\big)\equiv\big(p+(q+r)\big) \tag{12.7}$$

图12-19展示了对应各子表达式的真值表。而标号为E的最后一列就表示整个表达式。不难看出，对应E的每一行都具有值1，这说明表达式(12.7)是重言式。这样一来，只要我们看到形如$(p+q)+r$的表达式，就可以直接将其替换为$p+(q+r)$。请注意，p、q和r可以代表任何表达式，只要两边的p、q和r各自使用了相同的表达式，可以保持一致即可。

p	q	r	$p+q$	$(p+q)+r$	$q+r$	$p+(q+r)$	E
0	0	0	0	0	0	0	1
0	0	1	0	1	1	1	1
0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	1
1	0	0	1	1	0	1	1
1	0	1	1	1	1	1	1
1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1

图12-19　证明OR的结合律的真值表

12.7.1　替换原则

正如我们在示例12.18中指出的，当给出涉及某些特定命题变量的法则时，该法则不仅适用于字面上的那些变量，而且可以用任何表达式来替换各变量。根本原因在于，在我们对重言式的一个或多个变量进行替换后，重言式还是重言式。这一事实称为替换原则（substitution principle）。[①]当然，我们必须用同样的表达式替换多次出现的同一个变量。

✦ 示例 12.19

逻辑运算符AND的交换律可以通过证明逻辑表达式$pq\equiv qp$是重言式得到验证。要得到这一法则的一些实例，可以对该表达式进行替换。例如，可以用$r+s$替换p，并用$\bar{r}$替换q，得到等价式

$$(r+s)(\bar{r})\equiv(\bar{r})(r+s)$$

请注意，这里为每个被替换的表达式都加上了括号，以防因为运算符优先级约定而意外改变运算符的分组。在该情况中，$r+s$两边的括号是必要的，但$\bar{r}$两边的括号则可以省略掉。

还有其他的替换实例如下。可以用r替换p，而且不替换q，这样就得到$rq\equiv qr$。可以只留下p，把q替换为常量表达式1（TRUE），从而得到p AND $1\equiv1$ AND p。不过，我们用r代替式子中出现的第一个p，并用另一个不同的表达式$r+s$替换第二个p。也就是说，$rq\equiv q(r+s)$不是重言式（如果$s=q=1$且$r=0$，它的值就是0）。

如果考虑表达式树，就可以看到替换原则一直成立的原因了。想象一下对应某个重言式的表达式树，比如图12-20所示的对应示例12.19中重言式的表达式树。因为该表达式是重言式，所以我们知道，不管为处于叶子节点处的命题变量指定什么真值，根节点的值都为真（只要我们为标号为某给定变量的各个叶子节点指定相同的真值）。

① 不应该把替换原则与“以相等换相等”弄混。替换原则只适用于重言式，而在任何表达式中都能够以相等换相等。

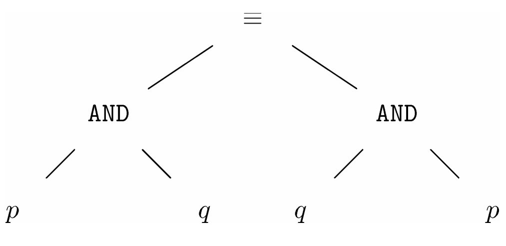


图12-20　对应重言式 $pq \equiv qp$ 的表达式树

现在假设用具有表达式树 T_p 的表达式替换p，并用具有表达式树 T_q 的表达式替换q，一般来说，我们会为重言式的每个变量选择一棵树，并用为该变量选择的树替换对应该变量的所有叶子节点。[①]这样就得到了一棵类似图12-21所示的新表达式树。当为新树的变量指定真值时，作为树 T_p 根节点的各个节点都有相同的值，因为任何这样的节点背后都执行了相同的求值步骤。

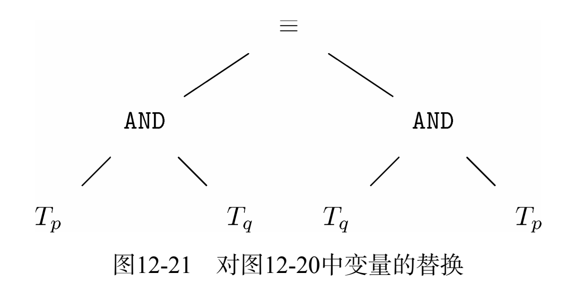


图12-21　对图12-20中变量的替换

一旦图12-21中 T_p 和 T_q 这样的树的根节点完成求值，就得到与图12-20所示的原有的树根节点处变量相同的值。也就是说，不管我们为出现的那些 T_p 算出什么样的值，它们一定是全部相同的，我们要取这个值，并将其指定给原有的树中标号为p的叶子节点。我们还要为q进行相同的处理，而且一般来说，要为出现在原有的树中的任何节点进行这一处理。因为原有的树表示重言式，所以可知为该树求值会在根节点得到值TRUE，而且新构造的树也能在根节点处生成值TRUE。因为不管为新树中的变量进行怎样的值替换，以上推理都能保持成立，所以可以得出结论：由新树表示的表达式也是重言式。

12.7.2　重言式问题

重言式问题就是测试某给定逻辑表达式是否等价于TRUE，也就是说，测试该表达式是否为重言式。有一种简单方式可以解决该问题。为该表达式构建真值表，其中每一行对应表达式中各变量的一种真值赋值。然后为该表达式的表达式树中各个内部节点创建一列，并按照合适的从下到上的次序，针对变量的各种真值赋值为各个节点求值。当且仅当对每种真值赋值而言整个表达式的值都是1（TRUE）时，该表达式是重言式。示例12.18就展示了这一过程。

12.7.3　重言式测试的运行时间

如果表达式有k个变量和n个运算符，那么这种真值表就有 2^k 行和n列需要填写。因此我们可以预期这种算法的简单实现要花 $O(2^k n)$ 的时间。这一时间对只有两三个变量的表达式来说并不长，即便是对20个变量来说，用计算机也只需要几分钟就能完成测试。不过，对30个变量而

① 作为特例，为某个变量x选择的树可能是标号为x的单个节点，就和没有对x进行替换一样。

言，有10亿行，就算是使用计算机，也几乎没法完成这一测试。这一结果是用到指数时间算法的典型下场。对较小的实例来说，一般看不出什么问题。但随着问题实例变大，突然间我们会发现，即使有着速度最快的计算机，也不可能在可以接受的时间内解决这个问题。

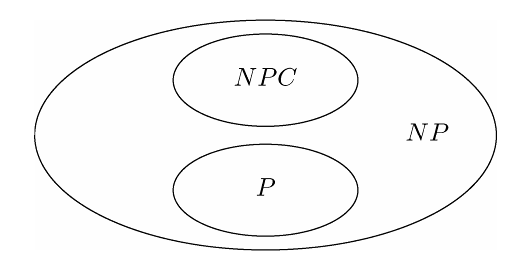


图12-22　*P*是可在多项式时间内解决的问题族，*NP*是可在非确定多项式时间内解决的问题族，*NPC*则是*NP*完全问题族

固有的难解问题

重言式问题“*E*是否为重言式”看似天生是指数时间的问题。也就是说，如果表达式*E*中有*k*个变量，所有已知解决重言式问题的算法的运行时间都是*k*的指数函数。

存在这样一类称为NP完全问题的问题，其中包含了很多重要的优化问题，而没人知道如何在少于指数时间的时间内解决这些问题。很多数学家与科学家经过长时间的艰难尝试，试着为这些问题中至少一个问题找到运行时间少于指数时间的算法，不过这样的算法还没被找到过，而很多人现在怀疑根本不存在这样的算法。

可满足性问题（satisfiability problem）就是一个经典的NP完全问题，这个问题是说“是否存在一种真值赋值让逻辑表达式为真？”可满足性问题与重言式问题有着密切的关系，而且就像重言式问题一样，对可满足性问题来说，也没有比循环经历所有可能的真值赋值好更多的解决方案了。

要么所有的NP完全问题都具有少于指数时间的解决方案，要么所有的NP完全问题都没有这样的解决方案。因此各NP完全问题看似需要指数时间这一事实让我们更加相信这些问题天生是指数时间的问题。有很强的迹象表明这种简单的可满足性测试就是最好的做法了。

顺便提一句，NP代表“非确定多项式”（Nondeterministic Polynomial）。粗略地讲，“非确定”就意味着“猜测正确的能力”，正如10.3节中讨论过的。如果为针对某个大小为*n*的实例的解决方案给出一次猜测，我们可以在多项式时间（也就是对某常数*c*而言的时间 n^c ）内验证该猜测是正确的，就说该问题能在“非确定多项式时间内解决”。

可满足性是这种问题的一个例子。如果为变量给出一组声明（或者说猜测）为可以使表达式*E*得到值1的真值赋值，我们可以将赋值代入操作数为*E*求值，并在至多为*E*的长度的二次方的时间内验证该表达式是否得到满足。

像可满足性问题这样可以通过猜测加上多项式时间的验证来“解决”的这类问题称为*NP*问题。有一些*NP*问题其实是相当简单的，不经过猜测就可以解决，而且只需要花输入长度的多项式的时间。不过，有很多*NP*问题被证实非常难，而这些问题就是NP完全问题。（不要把这里表示“这类问题中最难”的“完全”，与之前表达式“能表示每个布尔函数”的“运算符完全集”中的“完全”弄混了。）

在多项式时间内不通过猜测就可以解决的问题族通常称为*P*。图12-22展示了*P*、*NP*和NP完全问题之间的关系。如果任何NP完全问题在*P*中，那么 $P=NP$ ，我们会非常怀疑这种情况，因为所有已知的NP完全问题以及一些其他的*NP*问题，都不会出现在*P*中。没人相信重言式问题会在*NP*中，不过它的难度不低于*NP*中的任何问题（被称为*NP*难题），而且如果重言式问题在*P*中，那么有 $P=NP$ 。

12.7.4 习题

(1) 以下表达式中哪些是重言式？

(a) $pqr \to p+q$

(b) $\big((p \to q)(q \to r)\big) \to (p \to r)$

(c) $(p \to q) \to p$

(d) $\big(p \equiv (q+r)\big) \to (\bar{q} \to pr)$

(2) * 假设有一种为逻辑表达式解决重言式问题的算法，说明如何用这种算法实现下列目的。

(a) 确定两个表达式是否等价。

(b) 解决有关可满足性的问题（见上文附注栏“固有的难解问题”）。

12.8 逻辑表达式的一些代数法则

在本节中，我们将列举一些实用的重言式。在各种情况中，我们都只陈述法则，而将重言式的验证工作留给读者通过构造真值表来完成。

12.8.1 等价的法则

首先要从一些与等价如何起效有关的结论开始。大家应该注意到等价性在这里的双重身份。它是我们在逻辑表达式中使用的众多运算符之一。不过，它也是表示两个表达式“相等”并能互相替换的符号。因此形如 $E_1 \equiv E_2$ 这样的重言式表明了一些与 E_1 和 E_2 有关的信息，即利用“相等可以由相等替换”的原则，它们在更大的表达式中是可以相互替换的。

此外，我们可以利用等价证明其他的等价。如果有一列表达式 E_1、E_2、…、E_k ，满足每个表达式都能通过相等换相等的替换从前一个表达式得到，那么在用相同的真值赋值为这些表达式求值时，它们会得到相同的值。这样一来， $E_1 \equiv E_k$ 一定是重言式。

12.1 等价的自反性： $p \equiv p$ 。

正如我们要陈述的所有法则一样，替换原则是适用的，这样可以用任意表达式代替*p*。因此该法则表明任何表达式都是与自身等价的。

12.2 等价的交换律：$(p \equiv q) \equiv (q \equiv p)$ 。

非正式地讲，当且仅当*q*等价于*p*时有*p*等价于*q*。根据替换原则，如果任一表达式 E_1 与另一表达式 E_2 等价，那么 E_2 就等价于 E_1 。因此 E_1 和 E_2 是可以互相替换的。

12.3 等价的传递性：$((p \equiv q)\text{AND}(q \equiv r)) \to (p \equiv r)$ 。

非正式地讲，如果p等价于q，而且q等价于r，那么p就等价于r。这条法则具有一个重要的推论，如果我们得出 $E_1 \equiv E_2$ 和 $E_2 \equiv E_3$ 是重言式，那么 $E_1 \equiv E_3$ 也是重言式。

12.4 否定的等价：$(p \equiv q) \equiv (\bar{p} \equiv \bar{q})$。

当且仅当两个表达式的否定等价时，这两个表达式是等价的。

12.8.2 类似算术的法则

在算术运算符+、×和一元减号与逻辑运算符OR、AND和NOT之间存在一种类比。因此以下法则应该不会让大家感到意外。

12.5 AND运算的交换律：$pq \equiv qp$。

非正式地讲就是，只有在qp为真时，pq才为真。

12.6 AND运算的结合律：$p(qr) \equiv (pq)r$。

非正式地讲，要为3个变量（或表达式）的AND分组，既可以先取前两个变量（或表达式）的AND，也可以先取后两个变量（或表达式）的AND。此外，加上法则12.5，我们可以证明任意一系列命题或表达式的AND都可以按照我们的意愿随意排列和分组——结果都是相同的。

12.7 OR运算的交换律：$(p+q) \equiv (q+p)$。

12.8 OR运算的结合律：$\big(p+(q+r)\big) \equiv \big((p+q)+r\big)$。

这一法则和法则12.7表明了任何表达式集的OR都可以随意分组。

12.9 AND对OR的分配律：$p(q+r) \equiv (pq+pr)$。

也就是说，如果我们希望为p和两个命题或表达式的OR取AND，既可以先取OR，也可以对p与各表达式先取AND，得到的结果是相同的。

12.10 1（TRUE）是AND的单位元：$p \text{ AND } 0 \equiv p$。

请注意，$(1 \text{ AND } p) \equiv p$也是重言式。我们不需要说出它，因为它可由替换原则和之前的法则得出。也就是说，可以在12.5（AND的结合律）中用1替换p并用p替换q，从而得到重言式$(1 \text{ AND } p) \equiv (p \text{ AND } 1)$。然后，应用12.3（等价的传递性），就得到$(1 \text{ AND } p) \equiv p$。

12.11 0（FALSE）是OR的单位元：$(p \text{ OR } 1) \equiv p$。

同样地，可以利用与12.10如出一辙的论证，得出$(0 \text{ OR } p) \equiv p$。

12.12 0是AND的零元：$(p \text{ AND } 0) \equiv 0$。[①]

回想一下10.7节，运算符的零元是指这样一个常数，我们对该常数和任意值应用该运算符所得到的值都是该零元。请注意，在算术运算中，0是×的零元，但+是没有零元的。不过，我们会看到1是OR的零元。

12.13 双重否定的抵消：$(\text{NOT NOT } p) \equiv p$。

算术和逻辑运算符类比的利用

我们使用对应AND和OR的简写符号时，往往可以假装自己是在处理乘法和加法，正如我们在法则12.5到12.12中所使用的。这是种优势，因为我们对相应的算术运算法则是非常熟悉的。因此，大家应该能很快用$pr+ps+qr+qs$或$q(s+r)+(r+s)p$来替换$(p+q)(r+s)$。

更难也是更需要练习的部分就是应用那些与算术运算不相似的法则。比如德摩根律和OR对AND的分配律。例如，用$(p+r)(p+s)(q+r)(q+s)$替换$pq+rs$是可以的，但要看出这是通过3次应用OR对AND的分配律，并利用交换律和结合律得到的，需要费一些思量。

---

① 当然$(0 \text{ AND } p) \equiv 0$也成立，我们之后不会再提到那些结合律的结果。

12.8.3 AND和OR与加和乘的区别

还有很多法则表现出了AND和OR与算术运算符$\times$和+的区别，这里要列举一些。

12.14 OR对AND的分配律：$(p+qr)\equiv\big((p+q)(p+r)\big)$。

就像AND可以对OR分配那样，OR也可以对AND分配。请注意，相似的算术形式，$x+yz\equiv(x+y)(x+z)$一般而言是不成立的。

12.15 1是OR的零元：$(1\ \text{OR}\ p)\equiv 1$。

请注意，相似的算术形式$1+x=1$一般来说是不成立的。

12.16 AND的幂等性：$pp\equiv p$。

回想一下，当运算符应用到某相同值的两个副本时，得到的结果还是该值，就说该运算符是幂等的。

12.17 OR的幂等性：$p+p\equiv p$。

请注意，$\times$和+都不是幂等的。也就是说，一般来说$x\times x=x$和$x+x=x$都是不成立的。

12.18 吸收律。

这一法则有两个版本，取决于我们想消除的是多余的积还是多余的和。

(a) $(p+pq)\equiv p$

(b) $p(p+q)\equiv p$

请注意，如果在(a)中用任意文字积替换p，并用另一个文字积替换q，就可以说，在析取范式中，可以消除那些具有其他某个积所含文字之超集的积。较小的集合就被*吸收*到超集之中。在(b)部分中，我们对合取范式作出同样的说明，可以消除那些是其他某个和中文字之超集的和。

12.19 某些否定的消除。

(a) $p(\bar{p}+q)\equiv pq$

(b) $p+\bar{p}q\equiv p+q$

请注意，(b)就是我们在12.2节中解释莎莉的条件为何能替换山姆的条件时用到过的法则。

12.8.4 德摩根律

还有两条法则让我们可以把NOT压入AND和OR的表达式中，得到一个由各命题变量的否定组成的表达式。得到的表达式是应用到文字的AND-OR表达式。从直觉上讲，如果们为具有AND和OR的表达式取反，就可以把否定沿着表达式树向下压，随着该过程“翻转”运算符。也就是AND会变成OR，反之亦然。最后，否定到达叶子节点的位置，并停留在那里，除非它们遇到否定文字，在这种情况下，就要利用法则12.13消除两次否定。在构造新表达式时，一定要注意加上恰当的括号，因为在交换AND和OR时运算符的优先级改变了。

这些基本规则就叫“德摩根律”，它们是以下两个重言式。

12.20 德摩根律

(a) $\text{NOT}(pq)\equiv\bar{p}+\bar{q}$

(b) $\text{NOT}(p+q)\equiv\bar{p}\bar{q}$

(a)部分说明，只有在p和q之中至少有一个为假时，p和q才都不为真。而(b)部分说明，当且仅当p和q都为假时，p和q才都不为真。我们可以将这两条法则一般化，使其按照如下方式应用到任意数量的命题变量上。

(c) $\big(\text{NOT}(p_1p_2\cdots p_k)\big)\equiv(\bar{p}_1+\bar{p}_2+\cdots+\bar{p}_k)$

(d) $\big(\text{NOT}(p_1+p_2+\cdots+p_k)\big)\equiv(\bar{p}_1\bar{p}_2\cdots\bar{p}_k)$

例如，(d)说明当且仅当一系列表达式全为假时，它们才一个都不为真。

✦ 示例 12.20

我们已经在12.5和12.6节中了解到了如何为任意逻辑表达式构造析取范式。假设要从任意可以写成 $E_1+E_2+\cdots+E_k$，其中各 E_i 都是文字的AND的表达式E开始。就可以构造NOT E的合取范式，首先有

$$\text{NOT}(E_1+E_2+\cdots+E_k)$$

然后应用德摩根律(d)，得到

$$\big(\text{NOT}(E_1)\big)\big(\text{NOT}(E_2)\big)\cdots\big(\text{NOT}(E_k)\big) \tag{12.8}$$

现在设 E_i 是文字积 $\bar{X}_{i1}\bar{X}_{i2}\cdots\bar{X}_{ij_i}$，其中各$X$要么是变量，要么是变量的否定。那么我们可以对NOT(E_i)，将其变为

$$\bar{X}_{i1}+\bar{X}_{i2}+\cdots+\bar{X}_{ij_i}$$

如果某个文字X是否定变量，比方说是 $\bar{q}$，那么利用法则12.13，消除双重否定，$\bar{X}$ 就应该被替换成变量q本身。在进行所有的改变之后，式(12.8)就变成了文字和的积。

例如，$rs+\bar{r}\bar{s}$ 就是只有在 $r\equiv s$ 时才为真的析取范式，也就是说，它可以视作利用AND、OR和NOT对等价进行的定义。以下公式是上式的否定，只有在r和s不等价，也就是r和s刚好只有一个为真时才为真。

$$\text{NOT}(rs+\bar{r}\bar{s}) \tag{12.9}$$

现在对德摩根律(b)进行替换，用rs替换p，并用$\bar{r}\bar{s}$ 替换q。那么(b)的左边就成了式(12.9)，而根据替换原则可知，式(12.9)等价于对(b)进行相同替换后的右边，也就是

$$\text{NOT}(rs)\ \text{AND}\ \text{NOT}(\bar{r}\bar{s}) \tag{12.10}$$

现在我们可以应用(a)，其中用r替换p并用s替换q，将NOT (rs)转换成 $\bar{r}+\bar{s}$。同样，(a)告诉我们，NOT($\bar{r}\bar{s}$)与NOT($\bar{r}$)+NOT($\bar{s}$)是等价的。不过NOT($\bar{r}$)就等同于$\text{NOT}\big(\text{NOT}(r)\big)$，也就等价于$r$，因为双重否定是可以抵消的。同样NOT($\bar{s}$)也可以被$s$替代，因此式(12.10)等价于 $(\bar{r}+\bar{s})(r+s)$。这是表示“r和s刚好只有一个为真”的合取范式。粗略地说，它表示“r和s至少有一个为假，而且r和s至少有一个为真。”显然，这种情况只有在r和s中刚好有一个为真时才会发生。

12.8.5　对偶性原理

在审视本节所介绍的法则时，我们会注意到一个奇特的现象：这些等价性似乎都是成对出现的，只不过其中的AND和OR角色互换了而已。例如，法则12.19的(a)部分和(b)部分就是这样的一对，而法则12.9和12.14也是这样的一对，后者就是两条分配律。在涉及常数0和1时，它们也必须互换，就像在12.10和12.11这两条有关单位元的法则中那样。

在德摩根律中可以找到这一现象的解释。假设从重言式 $E_1\equiv E_2$ 开始，其中 E_1 和 E_2 都是涉及运算符AND、OR和NOT的表达式。根据法则12.4，有NOT(E_1)≡NOT(E_2)也是重言式。现在应用德摩根律把否定压过AND和OR。我们要做的，就是将每个AND“反转”为OR，反之亦然。而且我们会把否定下移到各操作数处。如果遇到NOT运算符，就直接把这个“移动的”NOT移到该

NOT运算符下方，直到遇到另一个AND或OR。例外就是当我们遇到否定的文字，比方说 $\overline{p}$ 时。然后，我们把这个移动的NOT与已经存在的那个结合起来，留下操作数p。作为特例，若移动的NOT遇到常数0或1，就要为该常数取否，也就是(NOT 0) ≡ 1和(NOT 1) ≡ 0。

✦ 示例 12.21

我们来考虑重言式12.19(b)。首先要为两边取否，这样就得到了图12-23a所示的树。然后把否定压过等价两边的OR，将它们变成AND，NOT符号就出现在两个OR的各参数之上，如图12-23b所示。新的NOT中有3个在变量之上，所以它们的移动就停止了。而在AND之上的那个会将该AND反转成OR，并使NOT出现在它的两个参数之上。这样右边的参数就成了NOT q，而左边的参数NOT p就成了NOT NOT p，也就是p。得到的树如图12-23c所示。

图12-23c的树表示表达式 $\overline{p}(p+\overline{q}) \equiv \overline{p}\,\overline{q}$ 。要让该表达式变成法则12.19(a)的形式，就必须为这些变量取否。也就是说，要用 $\overline{p}$ 替换p，并用 $\overline{q}$ 替换q。当消除双重否定之后，剩下的就刚好是法则12.19(a)。

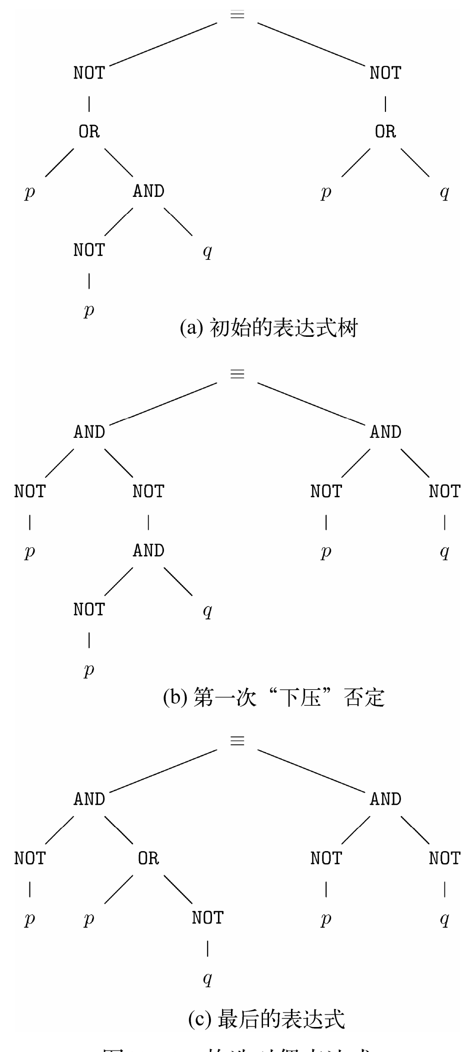


图12-23　构造对偶表达式

12.8.6　涉及蕴涵的法则

这里还有若干实用的重言式，给出了→运算符的属性。

12.21 $(p \to q)\text{AND}(q \to p) \equiv (p \equiv q)$。

也就是说，当前仅当两个表达式互相蕴涵时，它们是等价的。

12.22 $(p \equiv q) \to (p \to q)$。

两个表达式的等价表明其中一个蕴涵另一个。

12.23 蕴涵的传递性：$\big((p \to q)\text{AND}(q \to r)\big) \to (p \to r)$ 。

也就是说，如果p蕴涵q，而且q蕴涵r，那么有p蕴涵r。

12.24 可以把蕴涵用AND和OR表示出来，最简单的形式如下。

(a) $(p \to q) \equiv (\bar{p} + q)$ 。

我们会看到，很多情况下，要处理的表达式会形如“如果这个而且这个而且……，那么那个”。例如，Prolog语言和很多“人工智能”语言都依赖这种形式的“规则”。这些规则通常会写成$(p_1 p_2 \cdots p_n) \to q$ 。通过以下等价，它们可以只用AND和OR表示出来。

(b) $(p_1 p_2 \cdots p_n \to q) \equiv (\bar{p}_1 + \bar{p}_2 + \cdots + \bar{p}_n + q)$ 。

也就是说，只要q为真，或者这些p中有一个或多个为假，该等价的左边和右边就都为真，否则这两边都为假。

12.8.7　习题

(1) 通过构建真值表，验证法则12.1到12.24都是重言式。

(2) 可以用表达式替换重言式中的任何命题变量，并得到另一个重言式。在法则12.1到法则12.24这些重言式中，用$x+y$替换p，yz替换q，并用$\bar{x}$替换r，得到新的重言式。如果需要的话，不要忘了给新换上的表达式加上括号。

(3) 证明：

　(a) $p_1 + p_2 + \cdots + p_n$ 与 p_i 任意次序的和（逻辑OR）等价。

　(b) $p_1 p_2 p_n p_i$ 任意次序的积（逻辑AND）等价。

　提示：2.4节中为加法展示过相似的结果。

(4) * 利用本节给定的法则，把每一对表达式中的第一个表达式变形为第二个。为了减少工作量，在使用类似算术法则的法则12.5到12.13时，可以省略使用它们的步骤。例如，AND和OR的交换律和结合律是可以假定的。

　(a) 把 $pq + rs$ 变形为 $(p+r)(p+s)(q+r)(q+s)$ 。

　(b) 把 $pq + p\bar{q}r$ 变形为 $p(q+r)$ 。

　(c) 把 $pq + p\bar{q} + \bar{p}q + \bar{p}\bar{q}$ 变形为1（该变形需要用到12.9节介绍的法则12.25）。

　(d) 把 $pq \to r$ 变形为 $(q \to r) + (q \to r)$ 。

　(e) 把 $\text{NOT}(pq \to r)$ 变形为 $pq\bar{r}$ 。

(5) * 利用之前的法则证明吸收律12.18(a)和12.18(b)，也就是说明只使用法则12.1到12.17就可以把 $p + pq$ 变形为p，并可以把 $p(p+q)$ 变形为p。

(6) 应用德摩根律，将以下表达式变形为NOT只作用于命题变量（也就是NOT只出现在文字中）的表达式。

　(a) $\text{NOT}(pq + \bar{p}r)$

　(b) $\text{NOT}\big(\text{NOT}\, p + q\big(\text{NOT}(r + \bar{s})\big)\big)$

(7) * 利用基本法则12.20(a)和(b)，通过对k的归纳证明一般化的德摩根律12.20(c)和(d)。然后，通过描

述对应各表达式及其子表达式的真值表的样子，粗略验证这一一般化法则。

(8) * 找出本节中相互对偶的法则对。

(9) * 通过对n的归纳证明法则12.24(b)。

(10) * 通过描述对应表达式及其各子表达式具有 2^n 行的真值表，证明法则12.24(b)成立。

(11) 使用吸收律以及AND和OR的交换律和结合律，简化以下表达式

(a) $w\bar{x} + w\bar{x}y + \bar{z}\bar{x}w$

(b) $(w+\bar{x})(w+y+\bar{z})(\bar{w}+\bar{x}+\bar{y})(\bar{x})$

(12) * 通过给出一些特殊的数字使得类比的等式不成立，表明法则12.14到12.20的算术类比是不成立的。

(13) * 如果从那些只含AND、OR和NOT运算符的逻辑表达式开始，可以把所有的NOT向下压，直到NOT全部紧邻命题之上，也就是说，表达式是文字的AND和OR。证明我们能做到这一点。提示：只要看到NOT，要么它紧邻另一个NOT之上（这种情况下可以根据规则12.13抵消这两个NOT），要么它在命题之上（这种情况下命题就得到满足了），再或者它在AND和OR之上（这种情况下可以利用德摩根律将其压到下一层）。不过，想通过对诸如标号为NOT的节点高度之和这样显见的“大小”度量进行归纳，证明最终可以得到所有NOT都在命题之上的等价表达式，是不可能行得通的。原因在于，在利用德摩根律将NOT向下压时，它会变成NOT，这个和可能增加。为了证明最终可以得到所有NOT都在命题之上的等价表达式，需要找到一种合适的“大小”度量，在把NOT压到AND或OR之下的方向上应用德摩根律时，这个大小度量总是递减的。找到这样的大小度量，并证明该声明。

12.9 重言式及证明方法

在12.6到12.8这3节中，我们已经看到了逻辑的一个方面：它作为设计理论的用途。在12.6节中，我们看到如何利用卡诺图为给定的布尔函数设计表达式，而在第13章中我们会看到这种方法论是如何用到开关电路设计中的，而开关电路是构建计算机和其他数字设备的基础。12.7节和12.8节为我们介绍了重言式，它们可以用来简化表达式，因此在为给定布尔函数设计优质表达式时，重言式是另一种重要工具。

逻辑的第二个重要用途将在本节中得到体现。当人们推理或证明数学命题时，他们会用到很多技巧来推进自己的论证，这些技巧包括：

(1) 情况分析；

(2) 换质位法；

(3) 反证法；

(4) 归纳法。

本节中要定义这些技巧，展示它们各自是如何应用到证明中的。我们还会展示如何通过命题逻辑中的某些重言式来验证这些技巧。

12.9.1 排中律

首先要介绍一些表示与如何进行推理有关的基本事实的重言式。

12.25 排中律：$(p+\bar{p}) \equiv 1$是重言式。

也就是说，某事物要么为真，要么为假，不存在中间状态。

✦ 示例 12.22

作为法则12.25的应用，同时也利用到我们目前已经了解的若干其他法则，可以证明12.6节

中用过的法则 $(pq+\bar{p}q)\equiv q$ 。首先根据法则12.1，等价的自发性，用1 AND q替换p，就有

$$(1 \text{ AND } q)\equiv(1 \text{ AND } q)$$

接着，通过法则12.25，可以“以相等换相等”，用 $p+\bar{p}$ 替换上式左边的1，因此

$$\big((p+\bar{p})q\big)\equiv(1 \text{ AND } q)$$

是重言式。对该等价的右边使用法则12.10，用q替换1 AND q。然后对左边，我们使用12.9，AND对OR的分配律，进而利用法则12.5，AND的交换律，从而证实左边与 $pq+\bar{p}q$ 等价。因此有

$$(pq+\bar{p}q)\equiv q$$

这正是我们想要的。

将排中律一般化，就得到了名为“情况分析”（case analysis）的证明技巧，其中我们想证明某表达式E。我们会取另一个表达式F，及其否定NOT F，并证明F和NOT F都蕴涵E。因为F肯定要么为真要么为假，所以我们就得出了E。情况分析的正式依据是如下重言式。

12.26 *情况分析*：$\big((p\to q)\text{AND}(\bar{p}\to q)\big)\equiv q$ 。

也就是说，这两个实例是在p为真和p为假时发生的。如果q被两个实例蕴涵，那么q一定为真。我们把证明12.26可由12.25和其他已证明的法则得出这一任务留作本节习题。

12.27 $p\bar{p}\equiv 0$

命题及其否定不可能同时为真。这一法则在使用“反证法”时显得至关重要。我们很快就会在法则12.29中讨论这一证明技巧，而且在12.11介绍分解证明时也要提及。

12.9.2　换质位法

有时候我们想要证明 $p\to q$ 这样的蕴涵，却发现证明 $\bar{q}\to\bar{p}$ 这一被称为 $p\to q$ 的*质位变换命题*的等价表达式要更加简单。这一原则可以用如下法则公式化。

12.28 *质位变换法则*（contrapositive law）：$(p\to q)\equiv(\bar{q}\to\bar{p})$ 。

✦　示例 12.23

我们来考虑一个简单的例子，向大家展示可以如何利用质位变换法则。这个示例还表现了命题逻辑在证明过程中的局限。逻辑只能完成部分工作，允许我们在不参考命题本身含义的情况下对命题进行推理。不过，要得到完整的证明，通常还必须指定一些参数，让它们指代各项的含义。对本例来说，我们需要知道像“质数”、“奇数”和“大于”这样与整数有关的概念的含义是什么。

我们要考虑下面3个与正整数x有关的命题

a	“$x>2$”
b	“x是质数”
c	“x是奇数”

我们想要证明的定理就是 $ab\to c$ ，也就是

命题　“如果x大于2且是质数，那么x是奇数。”

首先要应用已经了解的一些法则，将表达式 $ab\to c$ 变形为更方便证明的等价表达式。首先，我们要利用法则12.28将其变成质位变换命题的形式 $\bar{c}\to \text{NOT}(ab)$ 。然后利用德摩根律12.20(a)将 $\text{NOT}(ab)$ 变形为 $\bar{a}+\bar{b}$ 。也就是说，可以把该定理变形为 $\bar{c}\to(\bar{a}+\bar{b})$ 。换句话说，需要证明

命题 “如果x不是奇数，那么它要么不大于2，要么不是质数。”
我们可以把“不是奇数”替换为“是偶数”，“不大于2”替换成“小于等于2”，并把“不是质数”替换为“是合数”。因此要证明的是

命题 “如果x是偶数，则要么有 $x<2$，要么有x是合数。”

现在已经将命题逻辑应用到极致了，接下来必须开始谈论这些项的含义了。如果x为偶数，那么对某个整数y而言有 $x=2y$，这也就是x为偶数的含义。因为在该证明中假设了x为正整数，所以y一定是大于等于1的。

现在可以使用情况分析了，分别考虑 $y=1$ 和 $y>1$ 这两种情况，因为我们论证过 $y \geqslant 1$，所以这是仅有的两种情况。如果 $y=1$，那么 $x=2$，这样就证明了 $x \leqslant 2$。如果 $y>1$，那么x是2和y这两个都大于1的整数的积，这就表示x是合数。因此我们证明了，如果x是偶数，那么要么有 $x \leqslant 2$（在 $y=1$ 情况下），要么有x是合数（在 $y>1$ 的情况下）。

12.9.3 反证法

我们经常不是“直接”证明表达式E，而是利用更简单的方式，首先假设NOT E，然后利用矛盾（也就是表达式FALSE）进行证明。这种证明的依据是以下重言式。

12.29 反证法：$(\bar{p} \to 0) \equiv p$。

粗略地讲，如果由 $\bar{p}$ 可以得出0，也就是得出FALSE或引起矛盾，就和证明了p是一样的。这一条法则其实是由其他法则得出的。如果用 $\bar{p}$ 替换法则12.24中的p，用0替代其中的q，就得到如下等价

$$(\bar{p} \to 0) \equiv \big(\text{NOT}(\bar{p})+0\big)$$

根据法则12.13，双重否定可以抵消，于是就可以把 $\text{NOT}(\bar{p})$ 替换为p，这样就有了

$$(\bar{p} \to 0) \equiv (p+0)$$

而法则12.11告诉我们，$(p+0) \equiv p$，进一步替换就得出了

$$(\bar{p} \to 0) \equiv p$$

✦ 示例 12.24

现在重新考虑一下示例12.23中的命题a、b、c，在这里例子中我们假设x是正整数，并分别断言 $x>2$、x是质数、x是奇数。我们想要证明定理 $ab \to c$，因此可以用该表达式替换法则12.29中的p。那么 $\bar{p} \to 0$ 就成了 $\big(\text{NOT}(ab \to c)\big) \to 0$。

如果对第一个蕴涵使用法则12.24，就得到

$$(\text{NOT}(\text{NOT}(ab)+c)) \to 0$$

对里层的NOT应用德摩根律就得到 $\big(\text{NOT}(\bar{a}+\bar{b}+c)\big) \to 0$。再次利用德摩根律，并两次利用法则12.13消除双重否定，就将该表达式变成了 $(ab\bar{c}) \to 0$。

这就是命题逻辑所能做到的极限了，现在必须进行与整数有关的推理。我们必须从a、b和$\bar{c}$开始，并得出矛盾。换句话说，首先要假设 $x>2$，x是质数，而且x是偶数，并一定要从这些假设中得出矛盾。

因为x是偶数，所以可以说对某个整数y而言有 $x=2y$。因为 $x>2$，就一定有 $y \geqslant 2$。不过这样一来，等于$2y$的x就是两个大于1的整数的积，也就是说x是个合数。因此就证明了x不是质数，也就是命题 $\bar{b}$。因为给定了b，也就是x是质数，现在又有了 $\bar{b}$，这样就有了 $b\bar{b}$，而根据法

则12.27，它是等于0，或者说为FALSE的。

于是证明了$(\text{NOT}(ab \to c)) \to 0$，根据法则12.29这就等价于$ab \to c$。这样也就完成了反证法证明。

12.9.4　等价于真

下一种证明方法让我们可以通过以相等换相等，直到表达式归约为1（TRUE），证明该表达式是重言式。

12.30 通过等价于真证明：$(p \equiv 1) \equiv p$。

✦　示例 12.25

表达式$rs \to r$表示，两个表达式的AND蕴涵了第一个表达式（而且根据AND的交换律，也蕴涵了第二个表达式）。可以通过以下一系列等价证明$rs \to r$是个重言式。

$$
\begin{aligned}
&rs \to r \\
&1) \equiv \text{NOT}(rs) + r \\
&2) \equiv (\bar{r} + \bar{s}) + r \\
&3) \equiv 1 + \bar{s} \\
&4) \equiv 1
\end{aligned}
$$

应用法则12.24，用AND和OR定义$\to$，得到(1)。应用德摩根律得出(2)。利用法则12.7和12.8，重新排列各项，然后根据法则12.25用1替代$r + \bar{r}$，就得到了(3)。最后，应用法则12.13，1是OR的零元，这样就有了(4)。

12.9.5　习题

(1) 证明法则12.25和12.27是相互对偶的。

(2) * 我们想证明定理“如果x是完全平方数而且x是偶数，那么x可以被4整除。”

- (a) 指定代表该定理中提到的3个有关x的条件的命题变量。
- (b) 把该定理用这些命题正式地表示出来。
- (c) 用命题变量的形式和口头描述的形式给出(b)小题得到命题的质位变换命题。
- (d) 证明(c)小题得到的命题。提示：要注意到，如果x不能被4整除，那么要么x是奇数，要么$x = 2y$且y是奇数。

(3) * 用反证法证明习题(2)中的定理。

(4) * 针对有关整数x的命题“如果x^3是奇数，那么x是奇数”，重复习题(2)和习题(3)（不过在习题(2)的(a)小题中只讨论了两种情况）。

(5) * 通过证明以下表达式等价于1（TRUE），证明它们是重言式。

- (a) $pq + r + \bar{q}\bar{r} + \bar{p}\bar{r}$
- (b) $p + \bar{q}\bar{r} + \bar{p}r + q\bar{r}$

(6) * 通过对之前已经证明的法则（的实例）进行以相等换相等，证明法则12.26：情况分析。

(7) * 将情况分析法则一般化为由k个命题变量定义情况的情形，这些变量在所有2^k个组合中可能为真或为假。那么验证$k = 2$的情况的重言式是什么？对一般的k来说呢？证明该重言式为何一定为真。

12.10　演绎

我们在12.6节到12.8节中看到了作为设计理论的逻辑，并在12.9节中看到了作为证明技巧的

逻辑。现在，我们要看看逻辑的第三面：逻辑在演绎中的使用。演绎就是可以构成完整证明的一系列命题。学习过平面几何的话就应该对演绎证明很熟悉，在演绎证明中我们从某些前提(“给定条件”)开始，并经过一系列步骤证明结论，其中每一步都是由前一个步骤经过有限次数的推理（称为推理规则）得到的。我们在本节中会解释演绎证明的构成，并给出若干示例。

不巧的是，为重言式找到演绎证明是很难的。正如我们在12.7节中提过的，这是个“固有的难解”问题，是属于NP困难问题一类的。因此要找到演绎证明，要么靠运气，要么就要穷举查找。在12.11节中，我们会讨论分解证明，虽然在最坏情况下它和其他技巧一样也必须花上指数时间，但它看起来是对寻找证明方法的一种好的探索。

12.10.1 演绎证明的构成

演绎的应用

除了作为数学证明的根本组成，演绎证明或者说形式证明在计算机科学中也有很多用途。应用之一是自动证明定理（automated theorem proving）。存在一些这样的系统：通过搜索从前提行进到结论的步骤序列，从而确定定理的证明过程。有些系统会自行查找证明，而另一些则会与用户进行交互，接受提示并填补构成证明的步骤序列中存在的小间隙。有人认为，虽然要让这样的系统投入实际使用还有很长的路要走，但它们最终可以用于证明程序的正确性。

演绎证明的第二个用途是用在与推导计算相关的编程语言中。举个简单的例子，机器人在寻找通过迷宫的路径时，可以把可能的状态用通道中心位置的有限集表示出来。我们可以绘制一幅图，其中的节点表示这些位置，而弧 $u \to v$ 就意味着机器人可以从位置u直接前移到位置v，因为u和v表示的是邻接的通道。

还可以把这些位置想象成命题，其中u代表“机器人可以到达位置u。”那么 $u \to v$ 就不仅能被解释为一条弧，还可以解释为一种逻辑蕴涵，也就是说“如果机器人可以到达u，那么它可以到达v。”（请注意这里的“双关”，箭头符号既可以表示弧，也可以表示蕴涵。）我们很自然地要问：“机器人从位置a可以到达哪些位置?”

如果取表达式a，以及所有对应邻接位置u和v的表达式 $u \to v$ 作为前提，看看可以从这些前提证明哪些命题变量x，就可以把该问题用演绎的形式表示出来。在这种情况下，我们并非真正需要像演绎证明这般强大的工具，因为正如在9.7节中讨论过的，深度优先搜索就能完成任务。不过，还有很多相关的情形，使用图论方法不是很有效率，但问题可以用演绎的形式表示，并得到合理的解决方案。

假设给定了某些逻辑表达式 E_1、E_2、…、E_k 作为前提，并希望得出形如另一个逻辑表达式E的结论。一般来说，结论和前提都不会是重言式，不过我们想要证明

$$(E_1 \text{ AND } E_2 \text{ AND} \cdots \text{AND } E_k) \to E \tag{12.11}$$

是个重言式。也就是说，想要证明，如果前提 E_1、E_2、…、E_k 为真，就能得到E为真。

一种证明(12.11)的方式就是为其构造真值表，并检验其中各行是否都是1，这是验证重言式的例行检验。不过，出于如下两个原因，这样可能并不足够。

(1) 正如上文提过的，如果表达式中存在太多的变量，重言式的检验就会变得非常棘手。

(2) 更重要的是，尽管重言式的验证对命题逻辑来说能起效，但它不能检验更复杂逻辑系统（比如第14章中将要讨论的谓词逻辑）中的重言式。

通常可以通过给出演绎证明来证明(12.11)是重言式。演绎证明是若干行组成的序列，每一行要么是给定的前提，要么是由之前的一行或多行根据推理规则构造出来的。如果最后一行是E，就说这是从E_1、E_2、⋯、E_k证明了E。

可以使用的推理规则有很多。唯一的要求就是，如果推理规则允许我们只要有表达式F_1、F_2、⋯、F_n是证明中的行，就可以把表达式F写为一行，就有

$$(F_1 \text{ AND } F_2 \text{ AND} \cdots \text{AND } F_n) \rightarrow \text{F}$$

一定是重言式。例如，

(a) 任何重言式都可以用作证明中的一行，而不管前面的行是什么。这一规则成立的原因在于，如果F是重言式，那么证明中0行的AND蕴涵了F。请注意，按照约定，0个表达式的AND是1，而当F为重言式时，$1 \rightarrow F$就是重言式。

(b) 肯定前件式假言推理（modus ponens）的规则是说，如果E和$E \rightarrow F$是证明中的行，就可以把F添加为证明的行。肯定前件式假言推理是由重言式$\big(p \text{ AND } (p \rightarrow q)\big) \rightarrow q$得出的，这里是用表达式$E$代替了$p$并用$F$代替了$q$。唯一的微妙之处在于，我们不需要$E$ AND $E \rightarrow F$这样一行，而是需要单独的两行，一行是E，一行是$E \rightarrow F$。

(c) 如果E和F是证明中的两行，那么可以添加一行E AND F。这样做可行的原因在于$(p \text{ AND } q) \rightarrow (p \text{ AND } q)$重言式，可以用任意表达式$E$替换$p$，用$F$替换$q$。

(d) 如果有E和$E \equiv F$这两行，那么可以添加一行F。可以这样做的原因类似于肯定前件式假言推理，是因为$E \equiv F$蕴涵$E \rightarrow F$。也就是说，$\big(p \text{ AND } (p \equiv q)\big) \rightarrow q$是重言式，而推理规则(d)是该重言式可替换出的实例。

无人喝彩的声音

我们常常需要理解为0个操作数应用运算符的极限情况，就像在推理规则(a)中所做的那样。我们断言，可以把0个表达式（或证明中的行）的AND当作具有真值1。这样的动机在于，除非F_1、F_2、⋯、F_n中有一个为假，否则$F_1 \text{ AND } F_2 \text{ AND} \cdots \text{AND } F_n$为真。但是如果$n = 0$，也就是一个$F$都没有，那么该表达式就不可能为假，因此很自然地将0个表达式的AND取为1。

我们要作出这样一个约定，只要对0个操作数应用运算符，得到的结果就应该是该运算符的单位元。因为可以预见0个表达式的OR是0，因为只要其中有一个表达式为真，多个表达式的OR就为真。同样，0个数字之和被取为0，而0个数字之积则被取为1。

✦ 示例 12.26

假设我们具有如下命题变量，具有表中所示的直觉含义。

r	“天在下雨。”
u	“乔伊带着伞。”
w	“乔伊被淋湿了。”

给定以下前提

$r \to u$	**“天在下雨，乔伊就会带着伞。”**
$u \to \bar{w}$	“乔伊带着伞，所以他没被淋湿。”
$\bar{r} \to \bar{w}$	“如果没下雨，乔伊是不会被淋湿的。”

现在要求证明 $\bar{w}$，也就是，乔伊绝不会被淋湿。从某种意义上讲，这个问题不值一提，因为大家可以验证

$$\big((r \to u)\text{AND}(u \to \bar{w})\text{AND}(\bar{r} \to \bar{w})\big) \to \bar{w}$$

是个重言式。不过，还是可以利用12.8节介绍的一些代数法则，以及刚刚讨论过的一些推理规则，从前提证明 $\bar{w}$。如果要处理形式比命题逻辑更为复杂的逻辑，或者是要处理涉及很多变量的逻辑表达式，就需要采取这种证明方式。图12-24展示了一种可能的证明方式，以及每个步骤进行下去的依据。

这种证明的大概思路是，利用情况分析，考虑天在下雨及没下雨这两种情况。根据第(5)行我们就证明了，如果天在下雨，那么乔伊不会被淋湿，而根据第(6)行，由给定的前提，说明如果没下雨，乔伊就不会被淋湿。第(7)到第(9)行结合了这两种情况，以得到我们想要的结论。

1)	$r \to u$	前提
2)	$u \to \bar{w}$	前提
3)	$(r \to u)$ AND $(u \to \bar{w})$	对(1)和(2)应用推理规则(c)
4)	$\big((r \to u)$ AND $(u \to \bar{w})\big) \to (r \to \bar{w})$	对法则(12.23)进行替换
5)	$r \to \bar{w}$	肯定前件式假言推理，以及(3)和(4)
6)	$\bar{r} \to \bar{w}$	前提
7)	$(r \to \bar{w})$ AND $(\bar{r} \to \bar{w})$	对(5)和(6)应用推理规则(c)
8)	$\big((r \to \bar{w})$ AND $(\bar{r} \to \bar{w})\big) \equiv \bar{w}$	对法则(12.26)进行替换
9)	$\bar{w}$	推理法则(d)，以及(7)和(8)

图12-24 演绎证明的示例

12.10.2 演绎证明“起作用”的原因

回想一下，演绎证明首先有前提 E_1、E_2、…、E_k 并要添加额外的行（也就是表达式），这些行都蕴涵自 E_1 AND E_2 AND…AND E_k。我们所添加的每一行都蕴涵自之前的0条或更多行，或者是某一行前提。我们可以通过对目前为止已经添加的行的数目进行归纳，来证明 E_1 AND E_2 AND…AND E_k 蕴涵了证明过程中的每一行。要完成这一任务，需要两个涉及语言的重言式族。第一个重言式族是从→的传递性一般化而来的。对任何n，有：

$$\big((p \to q_1) \text{ AND } (p \to q_2) \text{ AND}\cdots\text{AND } (p \to q_n) \text{ AND } \big((q_1q_2\cdots q_n) \to r\big)\big) \to (p - r) \quad (12.12)$$

也就是说，如果p蕴涵了各个 q_i，而且 q_i 一起蕴涵r，那么有p蕴涵r。

我们可以通过以下推理得出(12.12)是重言式。(12.12)为假的唯一可能就是 $p \to r$ 为假，而且左边那一串为真。不过 $p \to r$ 只能在p为真且r为假时为假，所以我们要假设p和 $\bar{r}$ 为真。然后必须证明(12.12)的左边为假。

如果(12.12)的左边为真，那么其中由AND连接的各个子表达式都为真。例如，$p \to q_1$ 为真。因为我们假设p为真，那么要让 $p \to q_1$ 为真，就只可能是 q_1 为真。同样，可以得出结论：q_2、…、q_n 都为真。因此 $q_1q_2\cdots q_n \to r$ 一定为假，因为我们假设r为假，而且刚刚发现所有的 q_i 都为真。

首先假设(12.12)为假，因此得到右边一定为真，所以p和$\bar{r}$都为真。然后可以得出，当p为真且r为假时，(12.12)的左边为假。但如果(12.12)的左边为假，则(12.12)本身为真，这样就得出了矛盾。因此(12.12)绝不可能为假，所以它是重言式。

要注意到，如果(12.12)中有$n=1$，就有了→的传递性的情况，也就是法则12.23。还有，如果$n=0$，那么(12.12)就成了$(1 \to r) \to r$，这是个重言式。回想一下，当$n=0$时，按约定$q_1q_2\cdots q_n$被取为AND的单位元，也就是1。

我们还需要一类重言式来证实可以为证明添加前提。它是对示例12.25中讨论过的重言式的一般化。我们声明，对任何满足$1 \leqslant i \leqslant m$的$m$和$i$来说，

$$(p_1p_2\cdots p_m) \to p_i \tag{12.13}$$

是重言式。也就是说，一个或多个命题的AND蕴涵它们之中的任何一个。

表达式(12.13)之所以是重言式，是因为唯一让它为假的可能就是左边为真且右边(p_i)为假。但如果p_i为假，那么p_i和其他p的AND必然为假，所以(12.13)的左边为假。但只要(12.13)的左边为假，它就为真。

现在可以证明，如果给定下列两个条件

(1) 前提E_1、E_2、⋯、E_k；

(2) 一组推理规则，满足只要这些推理规则允许我们写一行F，那么该行要么是E_i中的某一个，要么对某组之前的行F_1、F_2、⋯、F_n，存在重言式

$$(F_1 \text{ AND } F_2 \text{ AND}\cdots\text{AND } F_n) \to F$$

则对各行F，$(E_1 \text{ AND } E_2 \text{ AND}\cdots\text{AND } E_k) \to F$一定是重言式。要对添加到证明中的行的数量进行归纳。

依据。我们以0行的情况作为依据。这一命题成立，因为它表示的是与证明中每行F有关的信息，而且并没有这样的行需要讨论。也就是说，我们的归纳命题其实形如“如果F为证明的一行，那么……”，而且我们知道如果条件为假，那么这样的“如果–那么”命题就为真。

归纳。对归纳部分而言，假设对之前的各行G，有

$$(E_1 \text{ AND } E_2 \text{ AND}\cdots\text{AND } E_k) \to G$$

是重言式。设F是添加的下一行。就有两种情况。

演绎证明与等价证明

我们在12.8节和12.9节中看到的证明方式与12.10节研究的演绎证明有着不同的风格。不过，这两种证明都需要构建一系列的重言式，以得出所需的重言式。

在12.8节和12.9节中我们看到了等价证明，由一个重言式开始，通过各种替换得出另外的重言式。得到的所有重言式对某些表达式E和F而言具有$E \equiv F$的形式。这种证明风格会被用于三角学中，例如，在证明“三角恒等式”时会用到。

演绎证明也需要寻找重言式。唯一的区别在于，其中各个重言式都形如$E \to F$，其中E是前提的AND，而F是证明中我们实际要得出的行。事实上，我们不写出整个重言式只是一种表示上的便利，而非本质上的区别。我们也应该很熟悉这种证明方式，它常用于平面几何中的证明。

情况1：F是前提之一。那么$(E_1 \text{ AND } E_2 \text{ AND}\cdots\text{AND } E_k) \to F$是重言式，因为它源自(12.13)，是$m=k$，而且对$j = 1$、2、⋯、$k$，用$E_j$替换各个$p_j$得到的。

情况2：F是因为推理规则

$$(F_1 \text{ AND } F_2 \text{ AND} \cdots \text{AND } F_n) \to F$$

而被添加的，其中各个F_j都是之前各行中的某一行。根据归纳假设

$$(E_1 \text{ AND } E_2 \text{ AND} \cdots \text{AND } E_k) \to F_j$$

对各个j而言都是重言式。因此，如果用F_j替换(12.12)中的q_j，用

$$E_1 \text{ AND } E_2 \text{ AND} \cdots \text{AND } E_k$$

代替p，并用F代替r，我们就会知道，对表达式E和F中变量的真值进行任何替换，都会使(12.12)的左边为真。因为(12.12)是重言式，所以每一种真值赋值都会让右边为真。而这里的右边是$(E_1 \text{ AND } E_2 \text{ AND} \cdots \text{AND } E_k) \to F$。由此可以得出，该表达式对每种真值赋值都为真，也就是说，它是个重言式。

我们现在已经得出了归纳的结论，而且证明了对证明中的每行都有

$$(E_1 \text{ AND } E_2 \text{ AND} \cdots \text{AND } E_k) \to F$$

特别要说的是，若证明中最后的那行是我们的目标E，就知道$(E_1 \text{ AND } E_2 \text{ AND} \cdots \text{AND } E_k) \to E$。

12.10.3　习题

(1) * 从下列各小题给出的前提，分别给出对相应结论的证明。大家可以使用推理规则(a)到(d)。而对于重言式，只可以使用12.8节和12.9节中陈述过的法则，以及用“以相等换相等”的方式从这些法则的实例得到的重言式。

(a) 前提：$p \to q$，$p \to r$。结论：$p \to qr$。

(b) 前提：$p \to (q+r)$，$p \to (q+r)$。结论：$p \to q$。

(c) 前提：$p \to q$，$qr \to s$。结论：$qr \to s$。

(2) 说明以下内容为何是推理规则。如果$E \to F$是证明的一行，而且G是任何表达式，那么我们可以添加$E \to (F \text{ OR } G)$这样一行。

12.11　分解证明

正如我们在本章前面的内容中提过的，寻找证明是个困难问题，而且因为重言式问题看起来是天生的指数时间问题，所以没有通行的方式可以使寻找证明变得简单。不过，有很多已知的只针对“典型”重言式的技巧，看起来对找寻证明所需的探索工作有所帮助。在本节中我们就要研究一条实用的推理规则——分解（resolution），它可能是这些技巧中最基本的一条了。分解是基于以下重言式的。

$$\big((p+q)(\bar{p}+r)\big) \to (q+r) \tag{12.14}$$

这条推理规则的有效性是很容易验证的。想让它为假只有一种情况，就是$q+r$为假，而且左边的表达式为真。如果$q+r$为假，那么q和r都为假。假设p为真，则$\bar{p}$为假。那么$\bar{p}+r$为假，而(12.14)的左边就一定为假。同样，如果p为假，那么$p+q$为假，还是说明(12.14)的左边为假。因此，不可能让(12.14)的左边为真且右边为假，所以可以得出(12.14)是个重言式。

应用分解的一般方式是把前提转换成*子句*，这些子句是文字的和（逻辑OR）。我们可以把各个前提都转换成子句的积。而我们的证明要以这些子句作为证明中的行开始，这样做的原因是各子句都是“给定的”。然后应用分解规则构造新的行，而这些行总能证明是子句。也就是说，如果(12.14)中的q和r各自被任意文字和所替代，那么$q+r$也将是文字和。

在实际应用中，我们将通过删除重复来简化子句。也就是说，如果q和r都包含文字X，这种情况下就要从$q+r$中删除X的一个副本。之所以可以这样做，是因为有法则12.17、12.7和12.8，也就是OR的幂等性、交换律和结合律。一般而言，实用的看法是把子句看作文字的集合，而非文字的列表。结合律和交换律让我们可以按照任意方式排列文字，而幂等性则让我们可以消除重复。

还要消除那些含有互斥文字的子句。也就是说X和$\bar{X}$在一个子句中出现，那么利用法则12.25、12.7、12.8和12.15，该子句等价于1，就不需要在证明中包含该子句。也就是说，根据法则12.25，$(X+\bar{X})\equiv 1$，而且根据零元法则12.15，1与任何逻辑表达式求OR都等价于1。

✦ 示例 12.27

考虑子句$(a+\bar{b}+c)$和$(\bar{d}+a+b+e)$。我们可以让b扮演(12.14)中p的角色，那么q就是$\bar{d}+a+e$，而r就是$a+c$。请注意，我们已经重新进行了一些调整，把子句与(12.14)匹配起来。首先，第二个子句与(12.14)中的第一个子句$p+q$已经匹配，而第一个子句是与(12.14)的第二个子句匹配的。此外，扮演p的角色的变量一开始并未出现在我们的两个子句中，不过没关系，因为OR的交换律和结合律说明我们能够以任何次序重新排列子句。

如果我们的两个子句已经出现在证明中，则新子句$q+r$也可以作为证明中出现的行，这个新子句就是$(\bar{d}+a+e+a+c)$。可以消除多余的a，将该子句简化为$(\bar{d}+a+e+c)$。

再举个例子，考虑子句$(a+b)$和$(\bar{a}+\bar{b})$。如果a是(12.14)中的p，而q是b，r是$\bar{b}$，就得出新子句$(b+\bar{b})$。该子句等价于1，因此不需要生成。

12.11.1 把逻辑表达式变成合取范式

为了让分解起作用，需要把所有的前提以及结论，变成和的积的形式，也就是“合取范式”。可以采取的方法有若干种，最简单的可能就是以下这些。

(1) 首先，要消除除了AND、OR和NOT之外的任何运算符。我们根据法则12.21把$E\equiv F$替换为$(E\to F)(F\to E)$。然后，根据法则12.24，把$G\to H$替换为NOT(G)+(H)。NAND和NOR也很容易分别用NOT后加上AND或OR来替代。事实上，因为AND、OR和NOT是运算符的完全集，所以可知任何逻辑运算符，包括那些本书中没有介绍的，都可以用只涉及AND、OR和NOT的表达式替换。

(2) 接下来，应用德摩根律把所有的否定向下压，直到它们根据12.8节中的法则12.13被其他否定抵消，或是只应用到命题变量上。

(3) 现在要应用OR对AND的分配律，把所有的OR压到所有AND之下。得到是由OR结合的文字，然后是由AND结合的表达式，这就是合取范式表达式。

✦ 示例 12.28

我们来考虑以下表达式

$$p+\big(q\ \text{AND NOT}\big(r\ \text{AND}(s\to t)\big)\big)$$

请注意，为了均衡考虑简洁性和清晰性，我们在这里和以后的表达式中会混用简化符号和原始符号。

第(1)步要求把$s\to t$替换为$\bar{s}+t$，给出只含AND、OR、NOT的表达式

$$p+\left(q \text{ AND NOT}\left(r(\bar{s}+t)\right)\right)$$

在第(2)步中，必须利用德摩根律把第一个NOT向下压。在接下来的一系列步骤中NOT到达了命题变量。

$$p+\left(q\left(\bar{r}+\text{NOT}(\bar{s}+t)\right)\right)$$

$$p+\left(q\left(\bar{r}+(\text{NOT } \bar{s})(\bar{t})\right)\right)$$

$$p+\left(q\left(\bar{r}+(s\bar{t})\right)\right)$$

现在应用法则12.14，把第一个OR压到第一个AND以下。

$$(p+q)\left(p+\left(\bar{r}+(s\bar{t})\right)\right)$$

接下来要利用12.8节的法则12.8重新组合命题变量，从而把第二和第三个OR压到第二个AND之下。

$$(p+q)\left((p+\bar{r})+(s\bar{t})\right)$$

最后，再次利用法则12.14，所有的OR都在所有AND之下。得到的如下表达式就是合取范式。

$$(p+q)(p+\bar{r}+s)(p+\bar{r}+\bar{t})$$

12.11.2 利用分解的推理

我们现在看到了从前提E_1、E_2、…、E_k找到E的证明的一种方法，并了解了其大致脉络。把E和E_1、E_2、…、E_k分别转换为合取范式表达式F和F_1、F_2、…、F_k。我们要为一对对子句应用分解规则，因此要为证明添加新子句作为证明的行。然后，如果向证明中添加了F的所有子句，就证明了F，从而也证明了E。

✦ 示例 12.29

假设以表达式$(r \to u)\ (u \to \bar{w})\ (\bar{r} \to \bar{w})$作为前提。请注意，该表达式是示例12.26中使用过的前提的AND。[①]设像示例12.26中那样，所需的结论是$\bar{w}$。我们可以根据法则12.24，替换掉这些→，把前提转换成合取范式。至此，得到的结果已经是合取范式，不需要进行进一步的处理。所需的结论$\bar{w}$已经是合取范式，因为任何单个文字都是子句，而单个子句就是子句的积。因此我们一开始有子句

$$(\bar{r}+u)\ (\bar{u}+\bar{w})\ (r+\bar{w})$$

现在，假设要以r扮演p的角色，分解第一个和第三个子句，得到的子句就是$u+\bar{w}$。该子句可以与前提中的第二个子句一起被分解，以u代替p，得到子句$(\bar{w})$。因为该子句就是所需的子句，所以任务就完成了。图12-25把证明过程展示为一系列语句，其中每一行都是一条子句。

1)	$(\bar{r}+u)$	前提
2)	$(\bar{u}+\bar{w})$	前提
3)	$(r+\bar{w})$	前提
4)	$(u+\bar{w})$	(1)和(3)的分解
5)	$(\bar{w})$	(2)和(4)的分解

图12-25 $\bar{w}$的分解证明

① 大家现在应该已经看到，不管是写成很多条前提，还是把它们用AND连接起来写出一条前提，都是可以的。

分解为何有效

一般来说，找到证明需要组织起从前提到结论这一系列行的运气或技巧。大家现在可能已经注意到，尽管很容易验证12.10节和12.11节中给出的证明是有效证明，而完成那些需要寻找证明的习题就要困难得多。一般来说，猜测示例12.29中那样为了生成某一子句或某些子句而要执行的一系列分解，并不比寻找证明简单多少。

不过，如果把分解与反证法结合起来，就像在示例12.30中那样，就可以看到分解的魔力。因为我们的目标子句是0，是“最小的”子句，突然间就有了搜寻“方向”的概念。这就是说，可以试着逐步证明更小的子句，以期最终能证明0。当然，这样的探索并不能确保成功。有时候，我们在开始缩小子句并最终证明0之前必须证明某个非常大的子句。

其实，分解是针对命题演算的完全证明过程。只要$(E_1E_2\cdots E_k) \to E$是重言式，就可以从以字句形式表示的E_1、E_2、…、E_k和NOT E得出0。是的，这就是逻辑学家们为“完全”赋予的第三个含义。回想一下，其他两种含义分别是“能够表示任何逻辑函数的运算符集合”，以及“NP完全”中那样指“一类问题中最难的问题”。这里还是要说，存在这样的证明并不代表找到这样的证明很容易。

12.11.3　利用反证法的分解证明

将分解用作证明机制的一般方式与示例12.29中的情况多少有些区别。我们不是从前提开始并试着证明结论，而是从前提和结论的否定开始，试着得出不含文字的子句。这一子句的值为0，或者说FALSE。例如，如果我们有子句(p)和$(\bar{p})$，就能以$q = r = 0$应用(12.14)，得到子句0。

这种方式之所以有效，是因为12.9节中提到的法则12.19，或者说$(\bar{p} \to 0) \equiv p$。在这里，设p是要证明的命题：对某些前提E_1、E_2、…、E_k和结论E而言，有$(E_1E_2\cdots E_k) \to E$。那么$\bar{p}$就是$\text{NOT}\big((E_1E_2\cdots E_k) \to E\big)$，或者利用法则12.24，就是$\text{NOT}\big(\text{NOT}(E_1E_2\cdots E_k) + E\big)$。若干次应用德摩根律就可以得出$p$等价于$E_1E_2\cdots E_k\bar{E}$。因此，要证明$p$，可以改为证明$\bar{p} \to 0$，或者说是证明$(E_1E_2\cdots E_k\bar{E}) \to 0$。也就是说，我们证明了前提和结论的否定一起蕴涵着矛盾。

✦　示例 12.30

现在重新考虑示例12.29，不过这次要从3个前提子句和所需结论的否定——子句(w)——开始。0的分解证明如图12-26所示。利用反证法，可以得出这些前提蕴涵了所需的结论$\bar{w}$。

1)	$(\bar{r} + u)$	前提
2)	$(\bar{u} + \bar{w})$	前提
3)	$(r + \bar{w})$	前提
4)	(w)	结论的否定
5)	$(u + \bar{w})$	(1)和(3)的分解
6)	$(\bar{w})$	(2)和(5)的分解
7)	0	(4)和(6)的分解

图12-26　利用反证法的分解证明

12.11.4　习题

(1) 利用真值表，验证表达式(12.14)是重言式。

a	某人的血型为A型
b	某人的血型为B型
c	某人的血型为AB型
o	某人的血型为O型
t	某人的血样进行测试*T*的结果为阳性
s	某人的血样进行测试*S*的结果为阳性

图12-27 习题(2)的命题

(2) 设命题具有如图12-27给定的含义，写出表示如下概念的子句或子句之积。
  (a) 如果测试*T*结果为阳性，那么此人的血型为A型或AB型。
  (b) 如果测试*S*结果为阳性，那么此人的血型为B型或AB型。
  (c) 如果某人血型为A型，那么测试*T*的结果为阳性。
  (d) 如果某人血型为B型，那么测试*S*的结果为阳性。
  (e) 如果某人血型为AB型，那么测试*T*和测试*S*的结果都为阳性。提示：请注意，$(\bar{c}+st)$ 不是子句。
  (f) 一个人的血型可能是A、B、AB或O其中的一种。
(3) 利用分解，找到由习题(2)中可以得出的所有重要子句。大家应该忽略那些可以简化为1（TRUE）的无关紧要的子句，还要忽略那些文字是其他某个子句*D*文字真超集的子句*C*。
(4) 为12.10节的习题(1)给出使用了分解和反证法的证明。

12.12 小结

在本章中，我们已经看到了命题逻辑的要素，包括下列这些：
- 基本运算符，AND、OR、NOT、→、≡、NAND和NOR；
- 利用真值表表示逻辑表达式的含义，包括根据表达式构造真值表以及根据真值表构造表达式的算法；
- 诸多应用到逻辑运算符的代数法则中的一部分。

我们还谈论了作为设计理论的逻辑，具体如下：
- 卡诺图如何有助于我们为至多含4个变量的逻辑函数设计简单的表达式；
- 逻辑代数法则有时候是如何用来简化逻辑表达式的。

然后，我们看到逻辑可以帮我们表示和理解常见的证明技巧，比如：
- 利用情况分析进行证明；
- 利用换质位法进行证明；
- 利用矛盾进行证明（反证法）；
- 利用对真实的归约进行证明。

最后，我们研究了演绎法，也就是一行行证明语句的构造，注意其中几点：
- 存在大量推理规则，比如“肯定前件式假言推理”，让我们可以从证明中之前的行构造新一行；
- 通过把证明的行表示为文字和，并把这些和以实用的方式组合，分解技巧通常能帮助我们迅速找到证明；
- 不过，尚无已知算法可以保证在少于以表达式大小为指数的时间内找到表达式的证明；

- 此外，因为重言式问题是“NP困难问题”，所以不存在少于指数时间的解决该问题的算法这一观点是非常可信的。

12.13　参考文献

对逻辑推理的研究可以追溯到亚里士多德的时代。Boole [1854]研究出了命题代数，而布尔代数正是由此而来。

Lewis和 and Papadimitriou [1979]对逻辑进行了稍微高级一些的处理，Enderton [1972]和Mendelson [1987]是常见的数学逻辑处理，Manna和Waldinger [1990]从证明程序正确性的视角表现了逻辑这一主题。

Genesereth and Nilsson [1987]是从逻辑在人工智能中的应用这一视角来处理逻辑的。在该书中，大家可以看到更多与寻找证明的探索有关的内容，其中包括类似分解的技巧。把分解作为证明方法的原始论文是Robinson [1965]。

要了解更多难解问题的理论，请阅读Garey and Johnson [1979]。NP完全问题的概念是由Cook [1971]提出的，而论文Karp [1972]则明确了常见问题相应概念的重要性。

Boole, G. [1854]. *An Investigation of the Laws of Thought*, McMillan; reprinted by Dover Press, New York, in 1958.

Cook, S. A. [1971]. “The complexity of theorem proving procedures,” *Proc. Third Annual ACM Symposium on the Theory of Computing*, pp. 151–158.

Enderton, H. B. [1972]. *A Mathematical Introduction to Logic*, Academic Press, New York.

Garey, M. R. and D. S Johnson [1979]. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York.

Genesereth, M. R.and N. J. Nilsson [1987]. *Logical Foundations for Artificial Intelligence*, Morgan-Kaufmann, San Mateo, Calif.

Karp, R. M. [1972]. “Reducibility among combinatorial problems,” in *Complexity of Computer Computations* (R. E. Miller and J. W. Thatcher , eds), Plenum, New York, pp. 85–103.

Lewis, H. R. and C. H. Papadimitriou [1981]. *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, New Jersey.

Manna, Z. and R. Waldinger [1990]. *The Logical Basis for Computer Programming* (two Volumes), Addison-Wesley, Reading, Mass.

Mendelson, E. [1987]. *Introduction to Mathematical Logic*, Wadsworth and Brooks, Monterey, Calif.

Robinson, J. A. [1965]. “A machine-oriented logic based on the resolution principle,” *J. ACM* **12**:1, pp. 23–41.

第 13 章

利用逻辑设计计算机元件

在本章中我们会看到第12章中学习的命题逻辑可以应用到数字电子电路的设计中。这样的电路在每台计算机上都能找到，它们使用两种电压电平（“高”和“低”）表示二进制数值1和0。除了了解设计过程之外，我们还将看到算法设计技巧，比如“分治法”，也可以应用到硬件上。其实，务必意识到设计执行给定逻辑函数的数字电路的过程，从本质上讲与设计执行给定任务的计算机程序的过程是非常类似的。不过二者所使用的数据模型却差异明显，一般来说电路会被设计成并行（同时）处理很多事务，而一般的编程语言都是顺序（一次一步地）执行它们的步骤。不过，像模块化设计这样的通用程序设计技巧也是适用于电路的。

13.1 本章主要内容

本章涵盖了数字电路设计中的以下概念。

- 门的概念，门是执行某一逻辑运算的电子电路（13.2节）。
- 门如何被组织成电路（13.3节）。
- 某些被称为组合电路的电路，它们是逻辑表达式的电子等价物（13.4节）。
- 电路设计所受的物理约束，以及电路要快速产生答案所必须具备的属性（13.5节）。
- 两个有趣的电路示例：加法器和多路复用器。13.6节和13.7节展示了如何利用分治法为这两个问题设计执行迅速的电路。
- 存储单元是一种可以记住其输入的电路，而组合电路则不能记住它之前接收的输入（13.8节）。

13.2 门

门是具有一个或更多输入的电子设备，可以假设各输入要么是值0要么是值1。正如之前提过的，逻辑值0和1通常使用两个不同的电压电平表示，不过这种物理表示方法并不会对我们产生影响。门通常具有一路输出，它是输入的函数，而且它的值也是0或1。

每个门都会计算某个特定的布尔函数。大多数电子“技术”（制造电子电路的方法）有利于为某些布尔函数而不是其他布尔函数构建门。特别要说的是，AND门（与门）和OR门（或门）通常是很容易构建的，NOT门（非门，也称反相器）也是。虽然像13.5节中要讨论的，AND门和OR门可以具有任意数量的输入，但通常会对门所具有的输入加以实际的限制。如果AND门的所

有输入都是1，它的输出就是1，而如果它的输入中至少有一个0，则其输出为0。同样，如果OR门的输入中至少有一个是1，那么它的输出是1，如果所有输入都是0，则其输出为0。反相器（NOT门）只有一路输入，如果它的输入是0，那么它的输出为1，而如果其输入为1，则输出是0。

我们还发现，在大多数技术中NAND门（与非门）和NOR门（或非门）也是很容易实现的。只有NAND门的所有输入都是1才产生输出0，否则输出就是1。当NOR门的所有输入为0时，其输入是1，否则它的输入是0。比较难以通过电子方式实现的逻辑函数是等价，该函数接受两路输入x和y，而且如果x和y都是1或都是0，那么产生的输出就是1。而当x和y中刚好有一个是1时，输出就是0。不过，我们可以通过实现能够识别逻辑函数 $xy+\bar{x}\bar{y}$ 的电路，用AND门、OR门和NOT门构建等价电路。

表示我们提到的门的符号如图13-1所示。除了反相器（NOT门）之外，所示的每种门都具有两路输入。不过，通过添加额外的线，很容易给出两个以上的输入。单输入的AND门和OR门也是可行的，但它们没有什么实际作用，只是把它的输出传递给输出。单输入的NAND门和NOR门其实就是反相器。

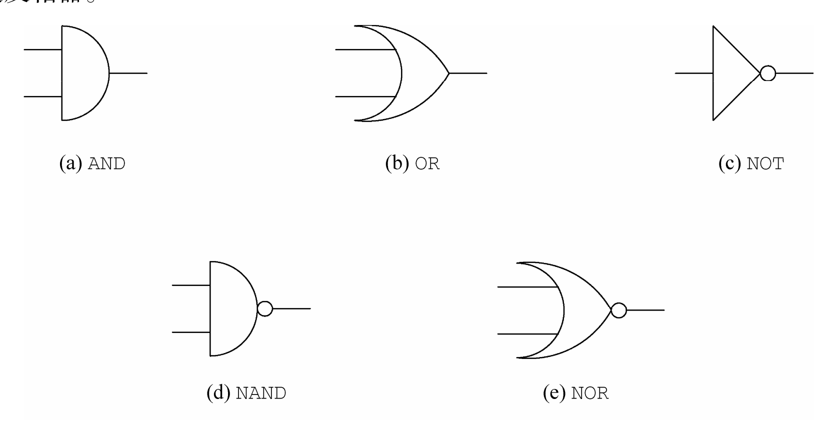


图13-1　表示门的符号

13.3　电路

通过连接某些门的输出与其他门的输入，就可以把门组合成电路。整个电路可能有一个或更多输入，每路输入都可能是电路中若干个门的输入。而一个或多个门的输出会被指定为电路的输出。如果存在多路输出，那么还必须为输出的门指定次序。

✦　示例 13.1

图13-2展示了产生输入x和y的等价函数作为输出z的电路。约定而言，我们把输入放在顶部。输入x和y都是提供给门A的，它是个AND门，所以当（且仅当）$x=y=1$时会产生输出1。此外，x和y会分别被NOT门反相，而且这些反相器的输出会被提供给AND门D。因此，当且仅当x和y都是0时，门D的输出是1。因为门A和门D的输出会提供给OR门E，我们可以看到当且仅当$x=y=1$或$x=y=0$时门E的输出是1。图13-3中的表给出了对应各门输出的逻辑表达式。

因此，当且仅当逻辑表达式$xy+\bar{x}\bar{y}$的值为1时，该电路的输出z（就是门E的输出）是1。因为该表达式等价于表达式$x\equiv y$，我们看到电路的输出是这两路输入的等价函数。

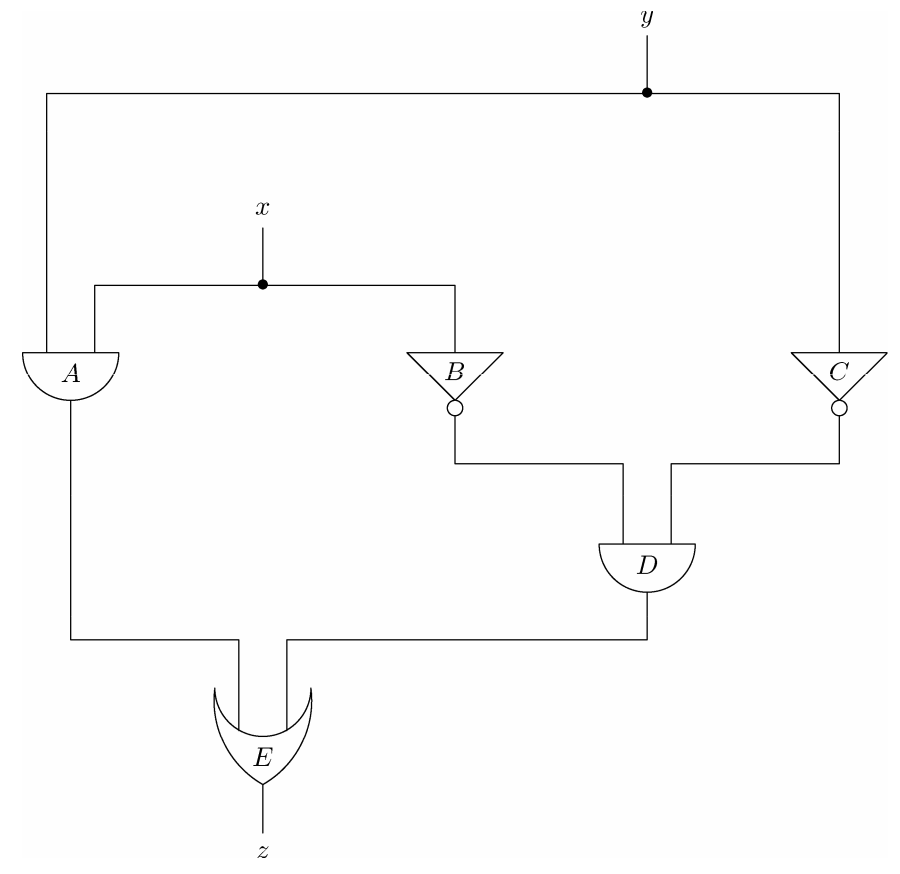


图13-2　等价电路：z是表达式 $x \equiv y$

门	门的输出
A	xy
B	$\bar{x}$
C	$\bar{y}$
D	$\bar{x}\,\bar{y}$
E	$xy+\bar{x}\bar{y}$

图13-3　图13-2中各门的输出

13.3.1　组合电路和时序电路

我们可以利用AND、OR和NOT这样的一系列逻辑运算符写出表达式，而这类表达式与可以用执行同一组运算符的门构建的电路之间存在着密切关系。在继续讲述之前，我们先把注意力集中在一类称为组合电路（combinational circuit）的重要电路上。这些电路是无环的，这种情况下门的输出不能到达其输入，即便是经过一系列中间门。

我们可以利用图的知识来精确定义想通过组合电路表示的含义。首先，画出图中节点对应电路中的门的有向图。如果门u的输出直接连接到门v的任何输入，就添加一条弧 $u \to v$ 。如果电路的图中没有环路，那么该电路就是组合电路，否则它就是时序电路。

✦ 示例 13.2

在图13-4中，我们看到根据图13-2所示电路画出的有向图。例如，存在弧 $A \to E$，因为门A的输出被连接到门E的输入。图13-4所示的图显然不含环路，其实，它是以E为根节点，方向倒置的树。因此，可以得出结论：图13-2所示的电路是组合电路。

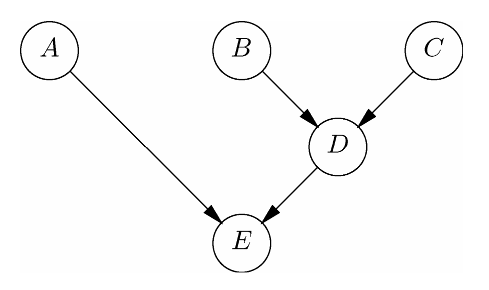
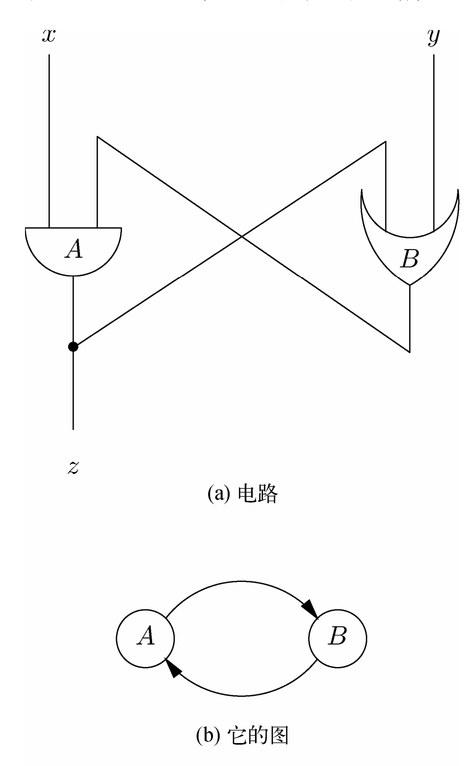


图13-4　根据图13-2中的电路构建的有向图

另一方面，考虑图13-5a的电路。在那幅图中，门A的输出是门B的输入，而门B的输出又是门A的输入。对应该电路的图如图13-5b所示，它显然具有环路，所以该电路是时序电路。



(a) 电路

(b) 它的图

图13-5　时序电路及其对应的图

假设该电路的输入x和y都是1，那么B的输出就肯定是1，这样AND门A的两路输入都是1。因此，该门会产生输出1。现在我们可以设输入y是0，而OR门B的输出仍然是1，因为它的另一路输入（来自A的输出的那路输入）是1。因此，A的输入仍然都是1，而它的输出也还是1。

不过，假设x变为0，而不管y是不是0。那么门A的输出以及电路的输出z一定是0。如果在过

去的某个时刻，x和y都是1，而且因此x（但y不一定）仍然是1，就可以把电路输出z描述为1。图13-6把若干输入值组合对应的输出表示成了时间的函数，低电平表示0，高电平表示1。

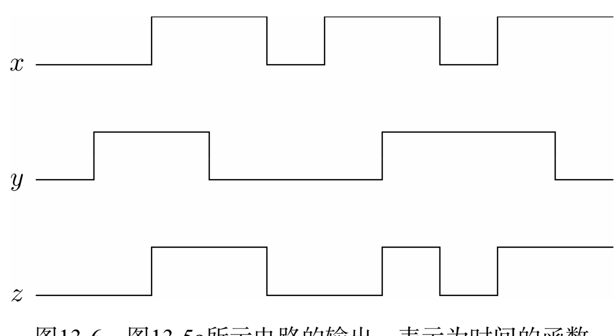


图13-6　图13-5a所示电路的输出，表示为时间的函数

时序电路和自动机

在第10章讨论过的确定有限自动机与时序电路之间存在密切关系。尽管该主题不在本书要讨论的范围之内，但给定任何确定自动机，我们都可以设计这样一个时序电路。只有当自动机的输入序列被接受时，该电路才输出1。更精确地讲，可能来自任意字符集合的自动机输入，一定要经过合适数量的逻辑输入进行编码，电路的k个逻辑输入最多可以编码2^k个字符。

我们将会在本章结尾部分简要讨论一下时序电路。正如我们在示例13.2中看到的，时序电路具备一种能力，可以记住与目前为止所见输入序列有关的重要事项，因此像主存和寄存器这样的计算机关键元件需要它们。另一方面，组合电路可以计算逻辑函数的值，但它们必须处理输入的单个设置，而且没法记住之前的输入被置为什么。不过，组合电路也是计算机的关键元件。组合电路为许多任务所需要，比如把数字相加，把指令解码成使计算机可以执行这些指令的电子信号等。在接下来的几节中，我们将把大多数精力放在组合电路的设计上。

13.3.2　习题

(1) 设计产生以下输出的电路，可以利用如图13-1所示的任何门。

- (a) 输入x和y的奇偶校验（或者说和mod2）函数，当且仅当x和y中刚好有一个是1时输出为1。
- (b) 输入w、x、y和z的多数（majority）函数，当且仅当输入中有3个或3个以上为1时输出是1。
- (c) 输入w、x、y和z的函数，只有在输入全是1或全不是1的情况下输出是1。
- (d) 12.4节习题(7)中讨论过的异或函数$\oplus$。

(2) * 假设图13-5a的电路被修改为门A和门B都是AND门，而且输入x和y一开始都是1。随着输入改变，在什么情况下输出会是1？

(3) * 如果两个门都是OR门，重复习题(2)。

13.4　逻辑表达式和电路

要构建输出（表示为其输入的函数）与给定逻辑表达式输出相同的电路是相当简单的。反过来，给定组合电路，我们也可以为电路的各路输出（表示为其输入的函数）找到相应的逻辑表达式。而正如我们在示例13.2中看到的，同样的做法并不适用于时序电路。

13.4.1　从表达式到电路

给定具有某些逻辑运算符的逻辑表达式，我们可以根据它构建一个组合电路，该电路使用具有相同运算符的门，而且可以识别相同的布尔函数。我们构造的电路总是具有树的形式，所以可以通过对表达式的表达式树进行结构归纳以构造电路。

依据。如果表达式树是单个节点，该表达式只能是一路输入，比方说x。而该表达式对应的“电路”就会是电路输入x本身。

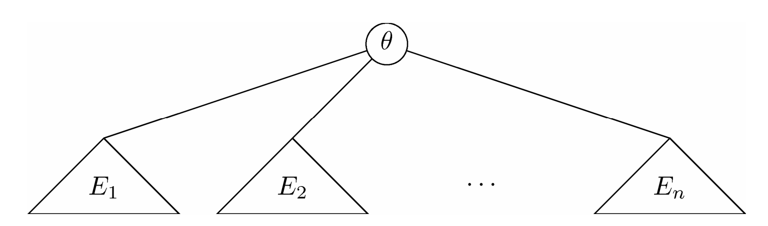


图13-7　对应表达式$\theta(E_1, E_2, \cdots, E_n)$的表达式树

归纳。而对归纳部分，假设所考虑的表达式树像图13-7这样。在根节点位置存在某个被称为θ的逻辑运算符，例如θ可以是AND或OR。根节点有n棵子树，而且要对各子树的结果应用运算符θ以产生整棵树的结果。

因为我们在进行结构归纳，所以可以假定归纳假设适用于子表达式。因此，存在对应表达式E_1的电路C_1、对应E_2的电路C_2，等等。

要为E构建电路，就要为运算符θ选择一个门，并为该门提供n路输入，其中各路输入按照次序分别是电路C_1、C_2、⋯、C_n的输出。而对应E的电路的输出来自刚介绍的θ门，电路构造如图13-8所示。

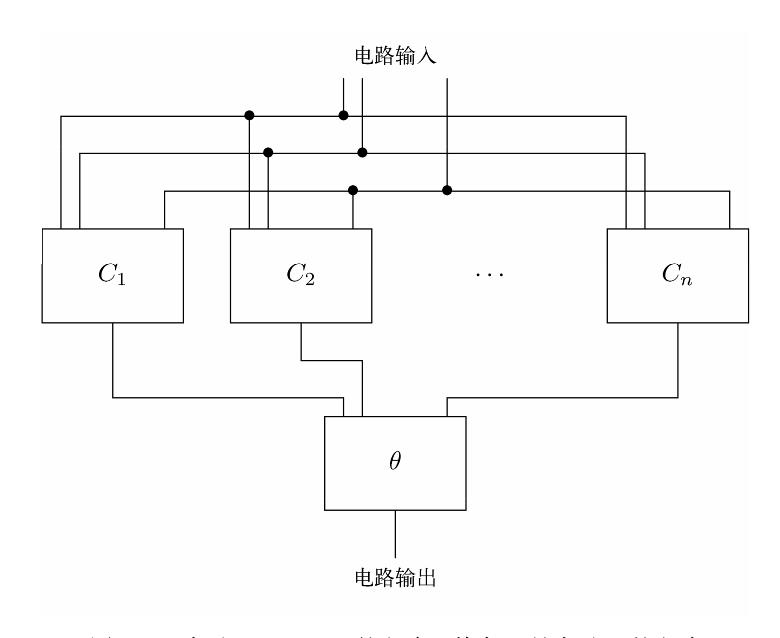


图13-8　表示$\theta(E_1, \cdots, E_n)$的电路，其中C_i是表示E_i的电路

我们所构建的电路是以显见的方式计算表达式的。不过，也可能存在产生相同输出函数但

所使用的门更少或电路层级更少的电路。例如，如果给定的表达式是$(x+y)z+(x+y)\overline{w}$，那么我们构建的电路就会出现两个识别相同表达式$x+y$的子电路。我们可以重新设计该电路，从而只使用一个这样的子电路，并为用到子表达式$x+y$的其他地方提供该子电路的输出。

要改进电路设计，还可以进行其他更为疯狂的变形。就像高效算法的设计一样，电路设计也是门艺术，而且我们还将在本章后续的内容中看到一些和电路设计有关的重要技巧。

13.4.2 从电路到逻辑表达式

现在来考虑一下相反方向的问题，为组合电路的输出构造逻辑表达式。因为我们知道组合电路的图是无环的，所以可以选定其节点（即电路中的门）的拓扑次序，而且具有如下属性：如果该次序中第i个门的输出被提供给第j个门的输入，那么i一定小于j。

✦ 示例 13.3

图13-2所示电路中的门可能的拓扑次序之一是*ABCDE*，而另一拓扑次序是*BCDAE*。不过*ABDCE*不是拓扑次序，因为门*C*要为门*D*提供输入，但该序列中*D*却出现在*C*之前。

要从电路构建表达式，就要使用归纳构造。这里将通过对i的归纳证明如下命题。

命题 $S(i)$。对拓扑次序中的前i个门来说，存在与这些门的输出对应的逻辑表达式。

依据。依据是$i=0$。因为要考虑的是0个门，就没什么要证明的，所以依据部分是成立的。

归纳。而对于归纳部分，要看看拓扑次序中的第i个门。假设门i的输入是I_1、I_2、$\cdots$、I_k。如果I_j是电路的输入x，那么令对应输入I_j的表达式E_j是x。如果I_j是其他某个门的输出，那么该门在该拓扑次序中一定先于第i个门，这表示我们已经为该门的输出构建了某个表达式E_j。设与门i相关联的运算符为θ，那么对应门i的表达式就是$\theta(E_1,E_2,\cdots,E_k)$。在$\theta$是约定使用中缀表示法的二元运算符的一般情况下，门$i$的表达式就可以写为$(E_1)\theta(E_2)$。虽然根据运算符的优先级这两个括号也有可能是不必要的，但为了安全起见还是用了括号。

✦ 示例 13.4

现在要利用门的拓扑次序*ABCDE*为图13-2所示的电路确定输出表达式。首先，我们要看看AND门*A*，它的两路输入来自电路的输入*x*和*y*，所以与*A*的输出对应的表达式就是*xy*。

门*B*是输入*x*的反相器，所以它的输出是$\overline{x}$。同样，门*C*的输出是$\overline{y}$。现在可以处理AND门*D*了，它的输入是*B*和*C*的输出。因此，对应门*D*输出的表达式为$\overline{x}\overline{y}$。最后，门E是OR门，它的输入是*A*和*D*的输出。因此要把这两个门的输出用OR运算符连接起来，从而得到表达式$xy+\overline{x}\overline{y}$，作为对应门*E*输出的表达式。因为*E*是电路唯一的输出门，所以该表达式也是电路的输出。回想一下，图13-2所示的电路整数是用来识别布尔函数$x\equiv y$的。很容易验证我们为门*E*得出的这个表达式与$x\equiv y$是等价的。

✦ 示例 13.5

在之前的例子中，我们的电路都只有一路输出，而且电路本身就构成了一棵树。但这些条件一般而言是不成立的。我们现在要介绍一个设计多输出电路的例子，而且其中一些门的输出会用作若干个门的输入。回想一下，第1章中讨论过用一位加法器构建一个计算二进制数字加法的电路。一位加法器电路有表示要相加的两个数字中某一特定位的两路输入*x*和*y*。除此之外，它还有第三路输入*c*，表示从其右侧相邻位置（低一位的位置）到该位的进位输入。而一位加法器会生成以下两位作为输出。

(1) 和值位z，当x、y和c中有奇数个是1时它的值为1。

(2) 进位输出位d，当x、y和c中有两个或三个是1时它的值为1。

在图13-9中，我们看到了对应一位加法器和值函数z与进位输出函数d的卡诺图。在8个可能的最小项中，其中有7个出现在对应z或d的函数中，而只有一个 xyc 是同时出现在两者之中。

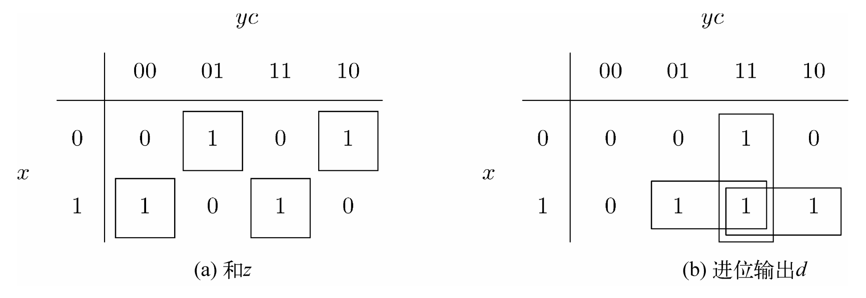


图13-9　对应和值函数和进位输出函数的卡诺图

图13-10展示了为一位加法器系统设计过的电路。首先要利用顶部的3个反相器为电路的输入反相。然后为输出中所需的各个最小项构建AND门。这些门编号为1到7，而且这些整数表明了它的输入中按次序有哪些是"为真"的电路输入x、y、c，以及有哪些是"互补的"输入，$\overline{x}$、$\overline{y}$、$\overline{c}$。也就是说，把这个整数写成3位的二进制数字，并用这几位按次序表示x、y和c。例如，门4，或者说是$(100)_2$，其输入x为真，而输入y和c是互补的，也就是说，它产生的输出表达式是 $x\overline{y}\overline{c}$。请注意，这里是没有门0的，因为每路输出都不需要最小项 $\overline{x}\overline{y}\overline{c}$。

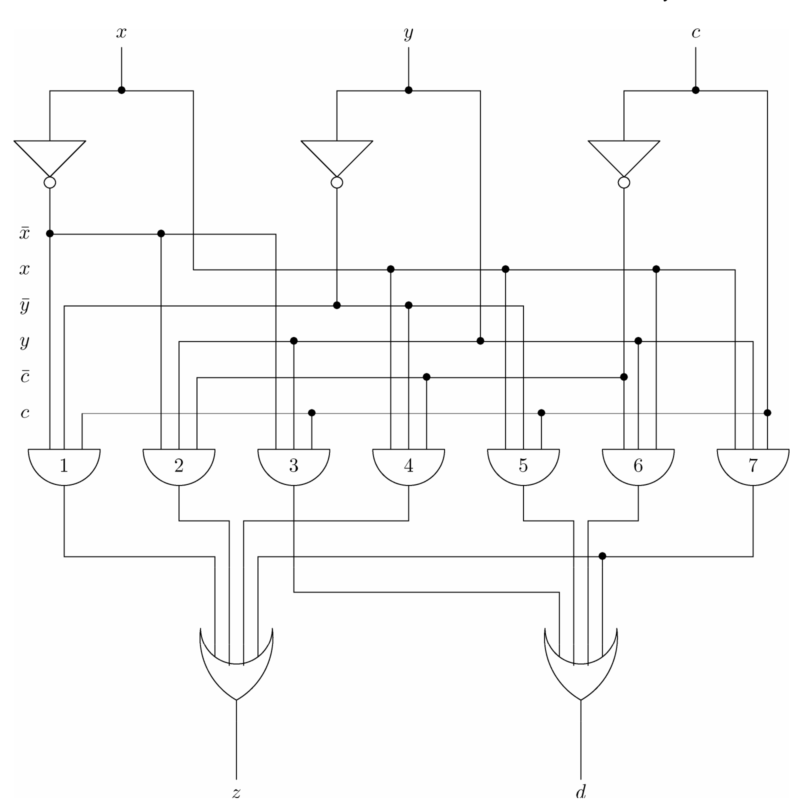


图13-10　一位加法器电路

最后，电路的输出z和d是由底部的OR门组合的。对应z的OR门的输入来自最小项中z为真的各个AND门的输出，而对应d的OR门的输入也是用类似方式选择的。

现在来为图13-10所示的电路求输出表达式。我们所使用的拓扑次序是反相器在前，接着是AND门1、2、⋯、7，以及最后的对应z和d的OR门。首先，这3个反相器的输出表达式显然是$\bar{x}$、$\bar{y}$和$\bar{c}$。然后，我们已经提过如何为这些AND门选择输入，以及各门输出对应的表达式与门编号的二进制表示之间是如何相关联的。因此，门1的输出表达式就是$\bar{x}\bar{y}c$。最后，OR门z的输出是对门1、2、4、7的输出表达式求OR，也就是

$$\bar{x}\bar{y}c + \bar{x}y\bar{c} + x\bar{y}\bar{c} + xyc$$

同样，对应d的OR门的输出是对门3、5、6、7的输出表达式求OR，即

$$\bar{x}yc + x\bar{y}c + xy\bar{c} + xyc$$

这里留一道习题给大家，证明该表达式与如下表达式是等价的。

$$yc + xc + xy$$

提示一下，如果我们从对应d的卡诺图着手，就能得到该表达式。

电路图的约定

如果电路像图13-10中所示的那样复杂时，就要拿出一项实用的约定来简化绘图。我们经常需要让“电线”（输出和输入之间的线）交叉，又不表示这些交叉的线是同一条线。因此，电路的标准约定是这样的：除非在线路相交的位置画上一个点，否则相交的线路不是连通的。例如，虽然从电路输入y出发的垂直线与标号x或$\bar{x}$的水平线有交叉，但它们是不连通的。它与标号为y的水平线是连通的，因为在相交的位置画上了点。

13.4.3 习题

(1) 为以下布尔函数设计电路。如果可以把由相同运算符连接的3个或更多操作数组合在一起，就不需要限制自己只使用2路输入的门。

  (a) $x + y + z$。提示：将该表达式视为OR(x, y, z)。

  (b) $xy + xz + yz$。

  (c) $x + (\overline{\bar{y}\bar{x}})(y + z)$。

(2) 为图13-11中的各电路计算对应各个门的逻辑表达式。电路输出的逻辑表达式又是什么？为电路(b)构造只使用AND、OR和NOT门的等价电路。

(3) 证明示例13.4和13.5中用到过的以下重言式。

  (a) $(xy + \bar{x}\bar{y}) \equiv (x \equiv y)$

  (b) $(\bar{x}yc + x\bar{y}c + xy\bar{c} + xyc) \equiv (yc + xc + xy)$

芯片

芯片通常含有若干结合起来可用于构建门的材料“层”。线路可以在任何一层布设，把门相互连接起来，不同层上的线路通常可以交叉而不互相影响。在1994年，芯片的“特征尺寸”（大体上就是电线的最小宽度）通常小于二分之一微米（1微米是0.001毫米，或者说大约0.00004英寸）。门可以构建在这若干微米见方的区域中。[1]

[1] 当今的芯片制造工艺已经达到纳米级的水平。——译者注

制造芯片的过程是很复杂的。例如，有一步是把薄薄一层光致抗蚀剂（photoresist）沉积在整个芯片上。然后要用到某层上所需功能的底片。通常用光或是电子束照射底片，被电子束直接照射到的那层会被蚀刻掉，只留下所需的电路。

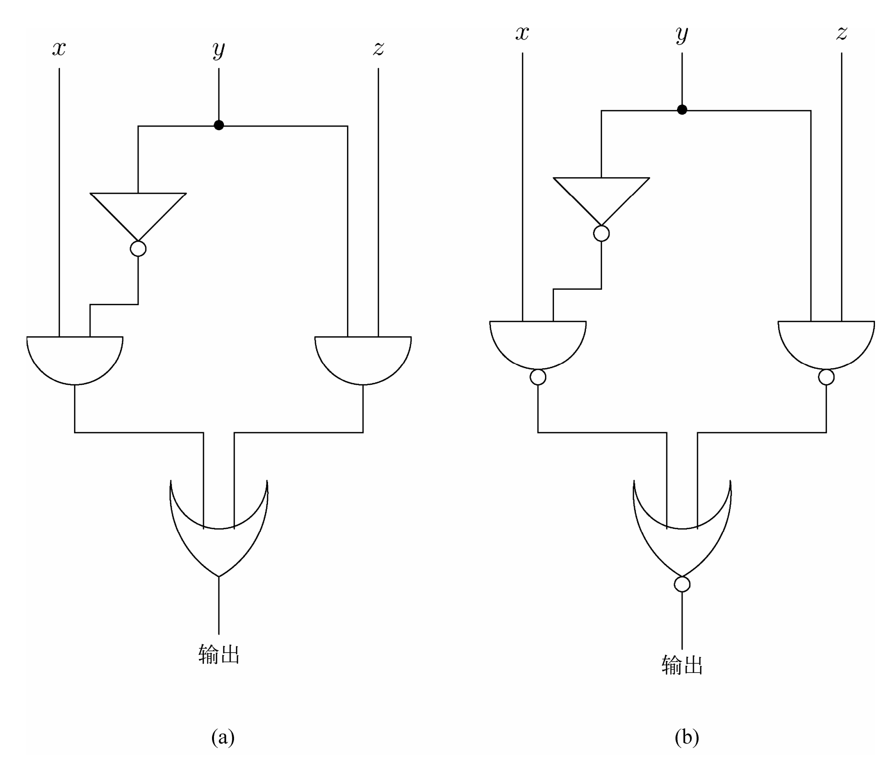


图13-11 习题(2)的电路

13.5 电路的一些物理限制

现在，很多电路都是以“芯片”或者说集成电路的形式构建的。大量的门，可能有多达数百万的门，以及连接这些门的线路，被构建在一块一厘米见方的半导体和金属材料上。构建集成电路的各种“技术”或者说方法都为设计高效电路之路施加了不少约束。例如，正如我们之前提到的，某些类型的门，比如AND、OR和NOT，就要比其他类型的门更容易构建。

13.5.1 电路速度

对每个门而言，在接收到输入的时间和发送出输出的时间之间都存在延迟。这一延迟可能只有几纳秒（1纳秒是10^{-9}秒），不过在复杂的电路，比如在计算机的中央处理器中，即便是在执行简单指令时，信息也会在很多层门之间传送。由于现代计算机能在远小于1微秒（即10^{-6}秒）的时间内执行指令，因此值在传递过程中必经之门的数量显然必须控制到最低限度。

因此，对组合电路而言，任何从输入到输出的路径上安放门的最大数量都是一种衡量性能

的标准，就类似于程序的运行时间那样。也就是说，如果我们希望电路能迅速计算输出，就必须把表示电路的图中最长路径的长度减少到最小。电路的*延迟*是最长路径上门的数量，也就是说，该最长路径的长度加1就是延迟。例如，图13-10所示的加法器延迟是3，因为从输入到输出的最长路径经过了一个反相器，然后是一个AND门，最后经过了一个OR门，这样的路径在该电路中有很多条。

请注意，和运行时间一样，电路延迟也只是一种“数量级”的量。不同的技术会让接受某个门的输入以产生该门输出的过程花费不同的时间。因此，如果有两个延迟分别为10和20的电路，那么可知，如果它们是用相同技术实现，而且其他因素也都相同，那么第一个电路所花的时间就是第二个电路的一半。不过，如果用更快的技术实现第二个电路，那么它有可能战胜用原技术实现的第一个电路。

13.5.2 大小限制

构建电路的开销大致上是与电路中门的数量成正比的，因此我们很乐意减少门的数量。此外，电路的大小也会影响到它的速度，而且小型电路往往运行得更快。一般来说，电路所具有的门越多，芯片上被占据掉的面积就越大。使用较大面积至少有如下两项负面影响。

(1) 如果面积较大，就需要较长的线路来连接相隔较远的门。线路越长，信号从线路一端传导到另一端所需的时间就越长。这种*传播延迟*（propagation delay）是电路中除了门“计算”输出所花时间之外的另一个延迟来源。

(2) 芯片的大小也是有限制的，因为芯片越大，就越有可能存在导致芯片不合格的瑕疵。如果把电路分散在若干芯片中，那么连接这些芯片的线路会带来严重的传播延迟。

结论就是，把电路中门的数量控制在较低水平会带来明显的好处。

13.5.3 扇入和扇出限制

电路设计中的第三个限制源自物理现实。如果门具有过多输入，或者门的输出连接到过多的输入，就会为此付出代价。门的输入数被称为*扇入*（fan-in），而门的输出所连接到的输入的数量就叫作*扇出*（fan-out）。尽管原则上讲扇入或扇出是存在限制的，但实际应用中，具有较大扇入和（或）扇出的门要比那些扇入和扇出较小的门更慢。因此，我们要试着在设计电路时对扇入和扇出加以限制。

✦ 示例 13.6

假设某计算的寄存器是32位的，而且我们想用电路实现COMPARE机器指令。必须构建的内容之一是测试寄存器是否全为0的电路。这一测试使用具有32路输入的OR门实现，每路输入对应寄存器的一位。输入为1就表示该寄存器存放的并不是0，而输出0就意味着它存放的是0。[①]如果要用1来表示问题“寄存器是否存放着0”的肯定回答，那么就要用反相器或者NOR门来为输出求补。

不过，扇入达到32一般来说远高于我们想要的值。假设要限制自己只使用扇入为2的门。这个限制可能太低了，不过这里只是为了举例说明问题。首先要问，如果需要计算n路输入的OR，那么需要多少个两输入的OR门？显然，每个两输入的门都会把两个值结合成一个值（它的输

① 严格地讲，这一结论只有在2的补码表示中才成立。在一些其他的表示法中，存在两种表示0的方法。例如，如果用符号幅度来表示，就只需要测试后31位是否为0。

出），因此会把我们计算n路输入的OR所需的输入数减1。在使用了$n-1$个门之后，我们会得到一个值，如果电路设计得当的话，这个值就是所有n个原来的值的OR。因此，至少需要31个门来计算x_1、x_2、…、x_{32}这32位的OR。

图13-12展示了实现这种OR的一种简单做法。在这种方法中，我们用左结合的方式为这些位分组。各个门的输出会提供给下一个门作为输入，该电路的图中有一条含31个门的路径，因此该电路的延迟是31。

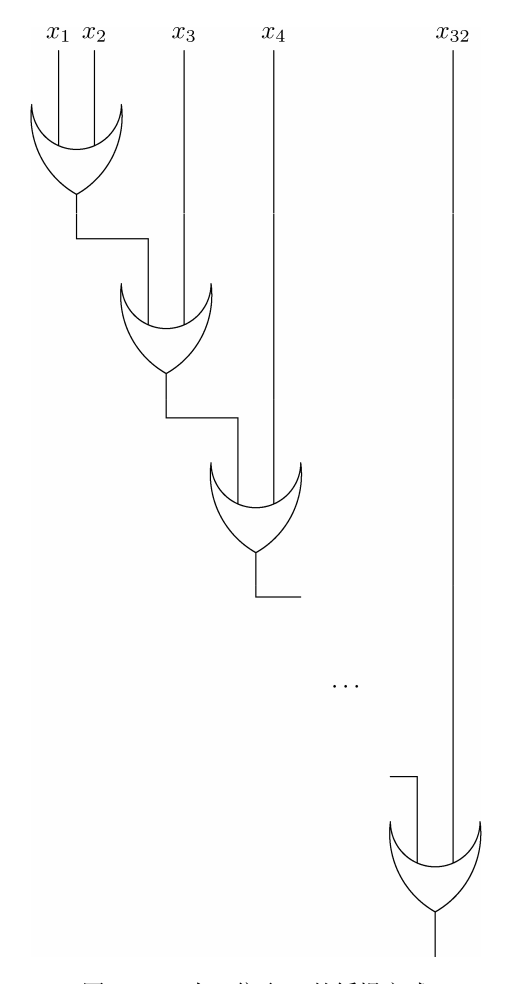


图13-12　为32位取OR的缓慢方式

图13-13展示了一种更好的方式。具有5层的完全二叉树使用了同样的31个门，不过延迟只有5。因此可以预期图13-13所示电路的运行速度是图13-12所示电路的6倍。其他影响速度的因素可能让这个倍数比6小，但是即便是对32位这样“小”的位数来说，这种聪明设计也明显要比之前的简单设计来得快。

如果不能马上“看出”使用完全二叉树作为电路的技巧，也可以利用分治范例得出图13-13所示的电路。也就是说，要取2^k位的OR，可以把这些位等分成各含2^{k-1}位的两组。这两组所对应的电路通过最终的OR门，如图13-14所示。当然，对应依据情况$k=1$（即两路输入）的电路不是用分治法得到的，而是使用单个两输入的OR门。

x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 … x_{29} x_{30} x_{31} x_{32}

图13-13　OR门的完全二叉树

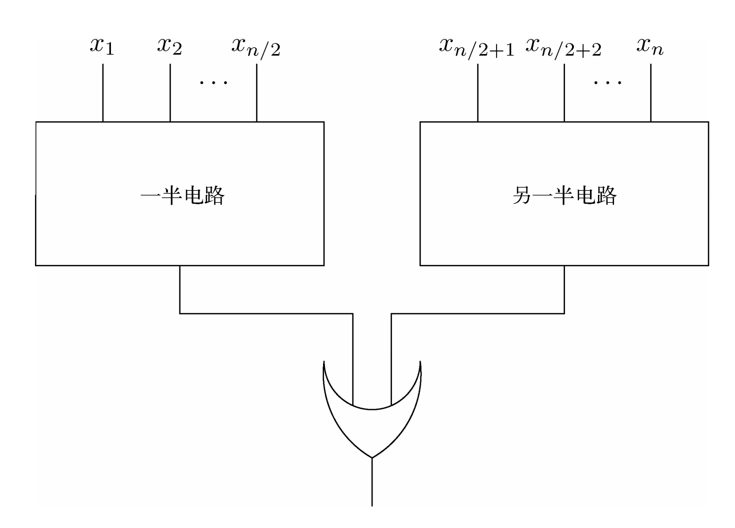


图13-14　把分治法用于电路设计

13.5.4　习题

(1) * 假设我们使用扇入为k的OR门，并且想为n路输入取OR，其中n是k的乘方。这种电路可能达到的最小延迟是多少？如果使用如图13-12所示朴素的“级联”电路，延迟会是多少？

(2) * 设计分治电路执行以下运算。每种电路的延迟各是多少？

(a) 给定输入x_1、x_2、…、x_n，当且仅当所有输入都是1时产生输出1。

(b) 给定输入x_1、x_2、…、x_n和y_1、y_2、…、y_n，当且仅当对i=1、2、…、n有$x_i = y_i$时，输出是1。提示：使用图13-2所示电路测试两路输入是否相等。

(3) * 即使输入数不是2的乘方，图13-14中的分治法也是起作用的。那么依据就一定要包括两输入或三输入的集合，三输入集合是由两个OR门处理的，假设我们要把门的扇入严格限制为2，就要用一个门的输出作为另一个门的一路输入。这种电路的延迟是多少，将其表示为输入数量的函数。

(4) 正选突击队准备就绪、意志坚定并能够出击。假设有n个突击队员，而且电路输入r_i、w_i和a_i分别表示第i个突击队员是否准备就绪、意志坚定并能够出击。只有当所有突击队员准备就绪、意志坚定并能够出击时，我们才派该突击队发动袭击。设计分治电路，表示我们能否派该突击队发动袭击。

(5) * 候补突击队（顺着习题(4)的思路）没有这么专业。如果各突击队员处在准备就绪、意志坚定或能够出击的状态，就派这支队伍发动袭击。其实，即便至多有一个突击队员既没有准备就绪，也不意志坚定，并且不能够出击，我们也派出这支队伍。使用与习题(4)一样的输入，设计能表示我们能否派候补突击队发动袭击的分治电路。

13.6　分治加法电路

将两个数字相加的电路是计算机的关键部分之一。尽管实际的微处理器电路所做的事更多，但我们这里要通过设计将两个非负整数相加的电路，研究该问题的本质。这一问题是一个相当有启发性的分治电路设计示例。

我们可以按照若干种连接方法中的某一种，用n个一位加法器构建n位数字的加法器。假设使用图13-10所示电路作为一位加法器电路。该电路的延迟是3，接近我们能达到的最低延迟。[①] 最简单的加法器构建方式是我们在1.3节中看到过的*行波进位加法器*。在该电路中，各一位加法器的输出都要称为下一个一位加法器的输入，所以把两个n位数字相加会带来3n的延迟。例如，如果是$n=32$的情况，那么该电路的延迟就是96。

13.6.1　递归加法电路

如果使用分治策略，设计处理n/2位的电路，并使用两个这样的电路以及其他一些补充电路构成n位加法器，就可以让设计出的加法器电路的延迟显著减少。在示例13.6中，我们讨论过使用两输入OR门为很多位取OR的分治电路。这是个特别简单的分治法应用示例，因为各个更小的电路执行的刚好是所需的功能（OR），而且子电路的输出组合是非常简单的（它们被提供给OR门）。这两个大小减半的电路可以同时（并行）处理它们的工作，所以它们的延迟不会叠加。

对加法器来说，我们需要完成一些更微妙的工作。比较简单的做法是使用同样的大小减半的加法器电路将左半部分的位（高序位）相加，并把右半部分的位（低序位）相加。不过，与n位OR的例子中可以独立地处理左半部分和右半部分不同的是，对加法器来说，似乎要在右半部

① 通过在全加器之外为所有输入求补，然后在全加器中计算进位和它的补数，就可以设计更为复杂但延迟为2的一位加法器电路。

分完成计算，并如图13-15所示把进位传递给左半部分的最右位之后，左半部分才可以开始计算。如果这样的话，我们会发现，这种所谓的“分治”电路其实就和行波进位加法器是一样的，而且根本没有改善延迟。

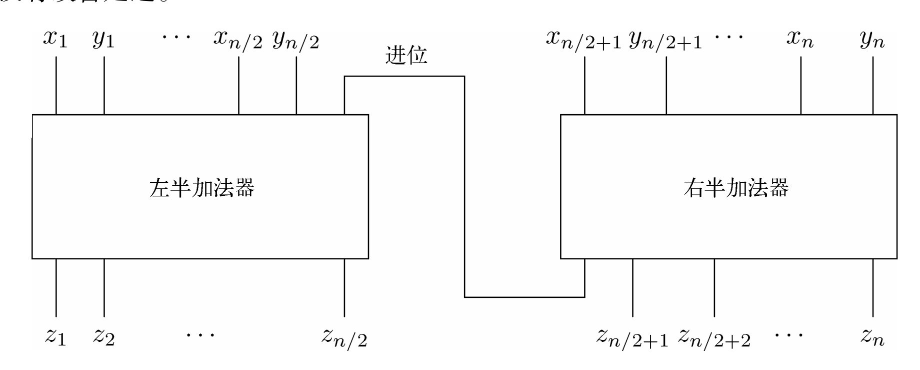


图13-15 无效的加法器分治设计

我们需要认识到的加法“诀窍”是，在要计算的不仅是和的条件下，我们可以在不知道右半部分进位输出的情况下计算左半部分。这里就需要回答两个问题。第一个，如果没有进位进入左半部分的最右位置，和会是多少，以及第二个，如果存在进位输入，和会是多少？[①]然后就可以让电路的左半部分和右半部分同时计算它们的答案。一般两个部分的计算都已完成，就可以弄清是否有从右半部分到左半部分的进位。这会告诉我们哪个结果是正确的，而且再经过三个单位的延迟，就可以为左边选出正确答案。因此，把n位相加的延迟只比把$n/2$位相加的延迟多3，这样就使电路的延迟是$3(1+\log_2 n)$。对$n=32$来说，这要比行波进位加法器好很多了，分治加法器的延迟是$3(1+\log_2 32)=3(1+5)=18$，而行波进位加法器的延迟是96。

更为准确地讲，我们将*n位加法器*定义为具有表示两个n位整数的输入x_1、x_2、⋯、x_n和y_1、y_2、⋯、y_n以及如下输出的电路。

(1) s_1、s_2、⋯、s_n，输入的n位和（不包括最左位置的进位输出，即不包括超出属于x_1和y_1的位置），假设最右的位置（x_n和y_n的位置）没有进位输入。

(2) t_1、t_2、⋯、t_n，输入的n位和，假设最右的位置有进位输入。

(3) p，*进位传送位*（carry-propagate bit），在假设最右位置有进位输入的情况下，如果最左位置存在进位输出，则它的值是1。

(4) g，*进位发生位*（carry-generate bit），即便最右位置没有进位输入，如果最左位置有进位输出，其值为1。

要注意到有$g \rightarrow p$，也就是说，如果g是1，则p一定是1。不过，g可以是0，而同时p仍是1。例如，如果x是1010⋯，而y是0101⋯，那么$g=0$，因为在没有进位输入时，求出的和全是1，而且最左位置没有进位输出。另一方面，如果最右位置有进位输入，那么后n位的和全部是0，而且最左位置有进位输出，因此$p=1$。

我们要为2的乘方n递归地构建n位加法器。

依据。考虑$n=1$的情况。这里有两路输入，x和y，而且需要利用以下逻辑表达式计算四路输出s、t、p和g：

① 请注意，“存在进位输入”表示进位输入是1，而“没有进位输入”意味着进位输入是0。

$$s = x\overline{y} + \overline{x}y$$
$$t = xy + \overline{x}\,\overline{y}$$
$$g = xy$$
$$p = x + y$$

要知道这些表达式为什么是正确的，首先要假设所考虑的这个位置没有进位输入。当x、y和进位输入中有奇数个1时就是1的和值位，在x和y中刚好有一个是1的情况下才是1。上述对应s的表达式显然具有这一属性。此外，在没有进位输入的情况下，只有在x和y都是1时才会有进位输出，这就解释了上面对应g的表达式。

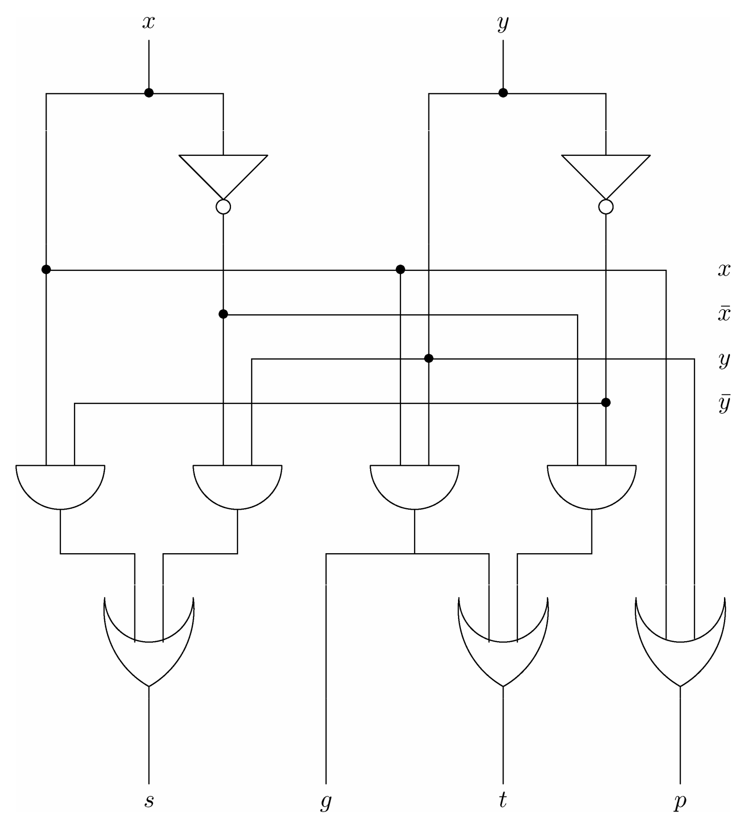


图13-16　依据情况：一位加法器

现在假设存在进位输入。那么对x、y和进位输入中有奇数个1的情况，就一定是x和y同为1或同不为1，这解释了对应t的表达式。还有，现在如果x和y中有一个是1或者两个全是1，就会有进位输出了，这就说明了对应p的表达式是正确的。对应该依据情况的电路如图13-16所示。它从思路上讲与图13-10所示的全加器是类似的，不过实际上它多少要简单一些，因为它只有两路输入。

归纳。归纳步骤如图13-17所示，其中用两个n位加法器构建了一个$2n$位加法器。$2n$位加法器是由两个n位加法器，加上两块图13-17所示的标号为`FIX`的电路组成的，其中`FIX`电路是用来处理以下两个问题的。

(1) 为$2n$位加法器计算进位传送位和进位发生位。

(2) 调整s和t的左半部分，以考虑是否具有从右半部分到左半部分的进位。

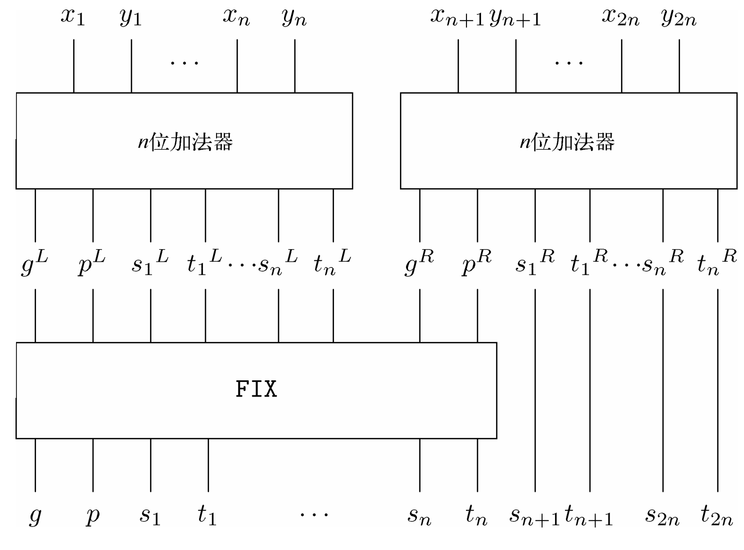


图13-17　分治加法器设计草图

首先，假设2*n*位加法器整个电路的右端有进位输入。然后，如果以下两个条件有任何一个成立，那么整个电路左端会有进位输出。

(a) 加法器的左半部分和右半部分都会传送进位，也就是说，$p^L p^R$ 为真。请注意，这一表达式包含了右半部分产生进位，而左半部分传送该进位的情况。那么 $p^L p^R$ 为真，但 $g^R \to p^R$，所以 $(p^L p^R + p^L p^R) \equiv p^L p^R$。

(b) 左半部分产生进位，也就是说，g^L 为真。在这种情况下，左端进位输出的出现并不取决于右端是否有进位输入，也不取决于右半部分是否产生了进位。

因为，对应2*n*位加法器的进位传送位*p*的表达式是

$$p = g^L + p^L p^R$$

现在假设2*n*位加法器的右端没有进位输入。那么只有出现了以下两种情况之一，2*n*位加法器的左端才会有进位输出。

(a) 右半部分产生了进位，而且左半部分传送了该进位；

(b) 左半部分产生了进位。

因此，对应*g*的逻辑表达式是

$$g = g^L + p^L g^R$$

现在把注意力转到这些 s_i 和 t_i 上。首先，右半部分的位与右侧*n*位加法器的输出相比没有改变，因为左半部分的出现不会对右半部分造成影响。因此，对 $i = 1$、2、⋯、n，有 $s_{n+i} = s_i^R$，而且 $t_{n+i} = t_i^R$。

不过，左半部分的位必须经过修改，从而把右半部分产生进位的情况考虑在内。首先，假设2*n*位加法器的右端没有进位。这种情况应该是由 s_i 告诉我们，从而可以为左半部分的 s_i（也就是 s_1、s_2、⋯、s_n）设计表达式。因为右半部分没有进位输入，所以只有在右半部分生成了进位的情况下，左半部分才有进位输入。因此，如果 g^R 为真，那么 $s_i = t_i^L$（因为 t_i^L 会告诉我们当左半部分有进位输入时会发生什么）。我们可以将其写为逻辑表达式

$$s_i = s_i^L \overline{g}^R + t_i^L g^R$$

其中，$i = 1$、2、⋯、n。

最后，要考虑当2*n*位加法器的右端有进位输入时会发生什么。现在可以按照如下方式解决左半部分t_i的值的问题。如果右半部分传送了进位的话，也就是如果$p^R=1$，左半部分就会有进位输入。因此，如果p^R为真，则t_i会接受t_i^L的值，而如果p^R为假，t_i就会接受s_i^L的值。写成逻辑表达式就是

$$t_i = s_i^L \overline{p}^R + t_i^L p^R$$

概括起来，图13-17中标有`FIX`的方框所表示的电路会计算如下表达式

$$p = g^L + p^L p^R$$

$$g = g^L + p^L g^R$$

$$s_i = s_i^L \overline{g}^R + t_i^L g^R\text{，其中}i = 1、2、\cdots、n。$$

$$t_i = s_i^L \overline{p}^R + t_i^L p^R\text{，其中}i = 1、2、\cdots、n。$$

这些表达式各自能被不超过3层的电路识别。例如，最后那个表达式只需要图13-18所示的电路。

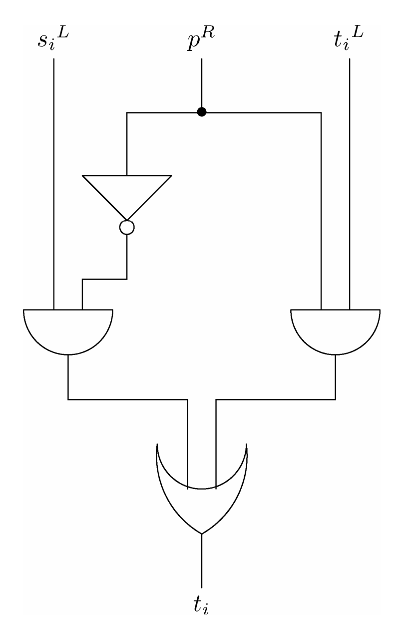


图13-18　`FIX`电路的一部分

13.6.2　分治加法器的延迟

设$D(\text{n})$是我们刚设计的*n*位加法器的延迟，可以按照以下方式写出表示*D*的递推关系。对依据情况$n=1$来说，查看图13-16中的依据电路，可以得出延迟是3。因此，$D(1)=3$。

现在要看看图13-17中电路的归纳构造。该电路的延迟是*n*位加法器电路的延迟，加上`FIX`电路的延迟。*n*位加法器的延迟是$D(n)$。而为`FIX`电路设计的各表达式都会带来一个不超过3层的简单电路。图13-18就是个典型的例子。因此，$D(2n)$要比$D(n)$多3。所以对应$D(n)$的递推关系是

$$D(1) = 3$$

$$D(2n) = D(n) + 3$$

对那些为2的乘方的位数来说，该递推关系的解有$D(1)=3$，$D(2)=6$，$D(4)=9$，$D(8)=12$，$D(16)=15$，$D(32)=18$，等等。对2的乘方*n*来说，该递推关系的解是

$$D(n) = 3(1 + \log_2 n)$$

大家可以用3.11节的方法检验。特别要说的是，请注意，对32位加法器而言，延迟18要远少于32位行波进位加法器的延迟96。

13.6.3 分治加法器使用的门的数量

我们还应该验证门的数量是否合理。设 $G(n)$ 是n位加法器电路使用的门的数量。依据是 $G(1) = 9$，数出图13-16所示电路中门的数量就可以得出这一数字。然后看到图13-17所示电路，也就是归纳情况，在两个n位加法器的子电路中有 $2G(n)$ 个门。除了这个量之外，还必须加上FIX电路中门的数量。可能要反转 g^R 和 P^R 一次，然后n个 s_i 和 t_i 各需要3个门（两个AND和一个OR）来计算，也就是总共要$6n$个门。在这个量之上，要加上为 g^R 和 P^R 设置的两个反相器，还必须加上计算g和p各自需要的两个门。因此FIX电路中门的总数量是$6n+6$。这样对应G的递推关系是

$$G(1) = 9$$
$$G(2n) = 2G(n) + 6n + 6$$

这里的函数还是只为2的乘方n定义。G的前6个值如图13-19中的表所示。对n=32，我们看到电路需要954个门。对2的乘方n来说，表示 $G(n)$ 的解析式是 $3n\log_2 n + 15n - 6$，大家可以利用3.11节中的技巧来证明该表达式是正确的。

n	$G(n)$
1	9
2	30
4	78
8	186
16	426
32	954

图13-19 多种n位加法器所使用的门的数量

事实上，如果所需要的只是32位加法器，完全可以用更少的门来实现电路。这样的话，可知在第32位的右边没有进位输入，因此在电路的最后阶段不需要计算p以及 t_1、t_2、…、t_{32} 的值。同样，右半部分的16位加法器也不需要计算它的进位传送和16个t的值，而右侧16位加法器右半部分的8位加法器不需要计算它的p和t，等等。

把分治加法器使用的门的数量与行波进位加法器使用的门的数量相比是很有意思的。我们在图13-10中设计的全加器电路使用了12个门。因此，n位行波进位加法器使用了12n个门，而对 $n = 32$ 来说，这个数字就是384。如果记得最右位的进位输入是0，还可以省掉一些门。

可以看到，对这种有意思的情况，也就是对 $n = 32$ 的情况而言，行波进位加法器尽管要慢很多，但使用的门的数量却不到分治加法器的一半。此外，后者的增长率 $O(n\log n)$ 要高于行波进位加法器的增长率 $O(n)$，所以随着n的增加，门数量的差别会越来越大。不过，这个比率只是 $O(\log n)$ 而已，所以门数量的差别并不严重。由于这两种电路所需时间（分别是 $O(n)$ 和 $O(\log n)$）的差别要更为明显，某种分治加法器几乎用在所有的现代计算机中。

13.6.4 习题

(1) 按照本节介绍的设计方法，画出把4位（bit）的数字相加的分治电路。

(2) 设计类似图13-18的电路，计算图13-17中加法器的其他输出，也就是p、g和那些 s_i。

(3) ** 设计接受十进制数字输入的电路，其中各位数字是4个给出与该十进制数字等价的二进制数字的输入表示的。而输出的是与之等价的二进制表示数字。大家可以假设数字（digit）的数量是2的乘方，并使用分治法。提示：左半部分的电路（高位数字）需要来自右半部分（低位数字）的哪些信息？

(4) * 证明，对2的乘方n而言，如下递推关系的解是$D(n)=3(1+\log_2 n)$。

$D(1)=3$

$D(2n)=D(n)+3$

(5) * 证明，对2的乘方n而言，如下递推关系的解是$G(n)=3n\log_2 n+15n-6$。

$G(1)=9$

$G(2n)=2G(n)+6n+6$

(6) ** 我们注意到，如果所需要的只是32位加法器，就不需要图13-19给出的全部954个门。原因在于，可以假设32位中最右的位置没有进位输入。那么实际上需要多少个门？

13.7　多路复用器的设计

多路复用器（multiplexer）通常简写为MUX，是一种常见的计算机电路，它接受d路控制输入，比方说是x_1、x_2、…、x_d，以及2^d路数据输入，比方说是y_0、y_1、…、y_{2^d-1}，如图13-20所示。MUX的输出等于特定的数据输入，输入$y_{(x_1x_2\cdots x_d)_2}$。也就是说，把控制输入当作0到2^d-1范围内的二进制整数，该整数是要传递给输出的数据输入的下标。

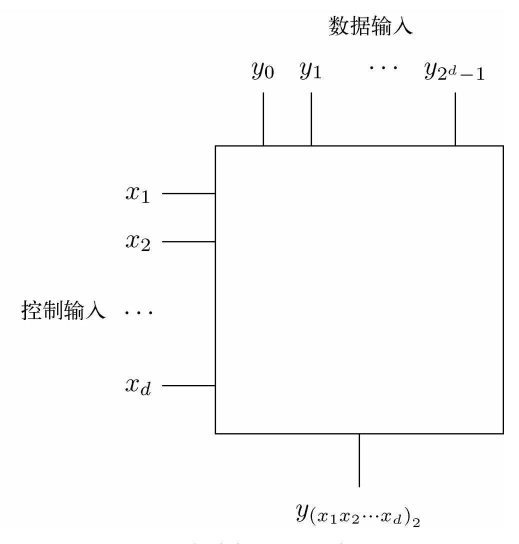


图13-20　多路复用器电路概要图

✦ 示例　13.7

分治加法器中计算s_i和t_i的电路就是$d=1$的多路复用器。例如对应s_i的公式是$s_i^L\overline{g}^R+t_i^Lg^R$，而它的电路概要图就如图13-21所示。这里，$g^R$所扮演是控制输入$x_1$的角色，$s_i^L$则是数据输入$y_0$，而$t_i^L$是数据输入$y_1$。

再举个例子，具有两个控制输入x_1和x_2，以及4个数据输入y_0、y_1、y_2和y_3的MUX的输出公式是

$$y_0\overline{x}_1\overline{x}_2+y_1\overline{x}_1x_2+y_2x_1\overline{x}_2+y_3x_1x_2$$

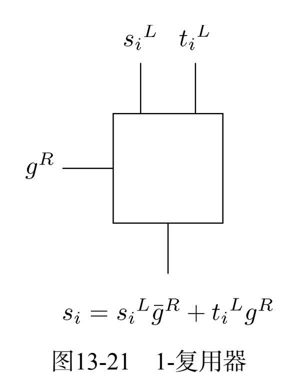


图13-21　1-复用器

这里对应各数据输入的都分别只有一项。而具有数据输入 y_i 的项也含有两个控制输入，要么是否定的，要么是非否定的。通过把i写为d位的二进制整数，我们可以推断出哪些控制输入是否定的。如果二进制的i第j个位置是0，那么 x_j 就是否定的，而如果第j个位置是1，就不为 x_j 取反。请注意，这一规则对任意数量d的控制输入都是有效的。

一种简单的多路复用器设计是使用具有3级门的电路。在第一级中，要计算各控制位的否定。接下来的一级是一行AND门。第i个门会把数据输入 y_i 与恰当的控制输入及取反控制输入组合结合起来。因此，除了控制位被置为i的二进制表示时第i个门的输出是 y_i，其他情况下该门的输出总是0。最后一级是一个OR门，它的输入来自上一级的各AND门。因为所有AND门中除了一个之外输出都是0，而这唯一一个输出不为0的AND门，比方说是第i个，它的输出是 y_i，所以电路的输出就等于 y_i。$d=2$ 时该电路的样子如图13-22所示。

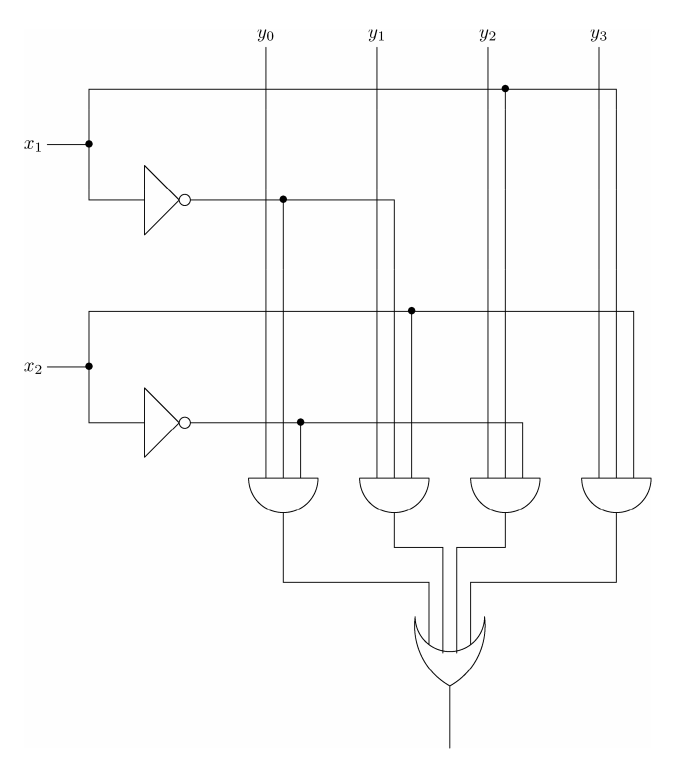


图13-22　$d=2$ 的多路复用器电路

13.7.1 分治多路复用器

图13-22所示电路的最大扇入是4，一般来说这是可以接受的。不过随着d逐渐变大，OR门的扇入2^d之大就变得不可接受了。即便是各自只有$d+1$路输入的AND门，也开始有着无法让人满意的大扇入。好在基于对控制位对半分割的分治法让我们可以用扇入至多为2的门来构建这样的电路。此外，假如要求所有电路都是用具有相同扇入限制的门构建的，这一电路使用的门就会少很多，而且几乎和图13-22所示的一般电路一样快。

我们把具有d路控制输入和2^d路数据输入的多路复用器称为d-MUX，那么这类多路复用器电路的归纳构建如下所述。

依据。依据是$d=1$时的多路复用器电路，也就是1-MUX，如图13-23所示。它由4个扇入被限制为2的门组成。

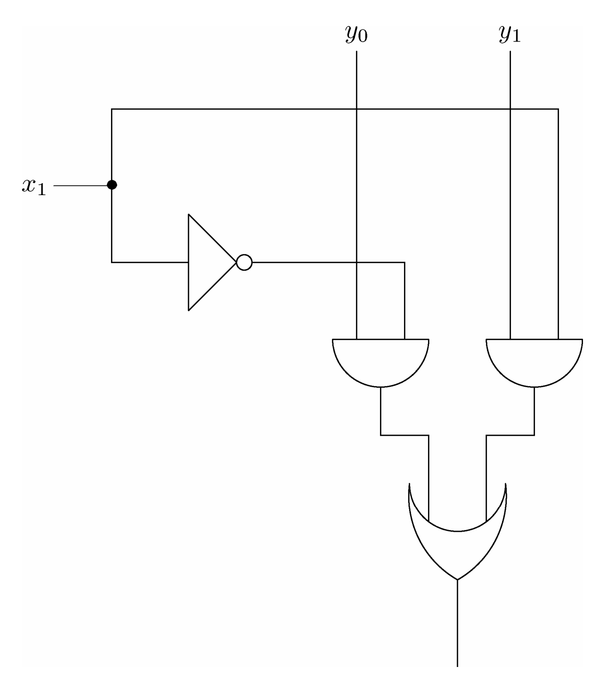


图13-23 依据电路，$d=1$时的多路复用器

归纳。归纳是由图13-24所示电路进行的，它是用2^d+1个d-MUX构建了一个$2d$-MUX。请注意，尽管控制输入的数量只是翻倍，数据输入的数量却变为之前的平方，因为$2^{2d}=(2^d)^2$。

假设$2d$-MUX的控制输入需要数据输入y_i，也就是

$$i=(x_1x_2\cdots x_{2d})_2$$

图13-24中顶部那行d-MUX接受一组从某个y_j开始的2^d路数据输入，这里j是2^d的倍数。因此，如果用低序的d个控制位x_{d+1}、⋯、x_{2d}来控制各个d-MUX，所选择的输入就是各组中的第k个（各组中最左侧的输入被记为输入0），其中

$$k=(x_{d+1}\cdots x_{2d})_2$$

也就是说，k是由低序那一半中的位表示的整数。

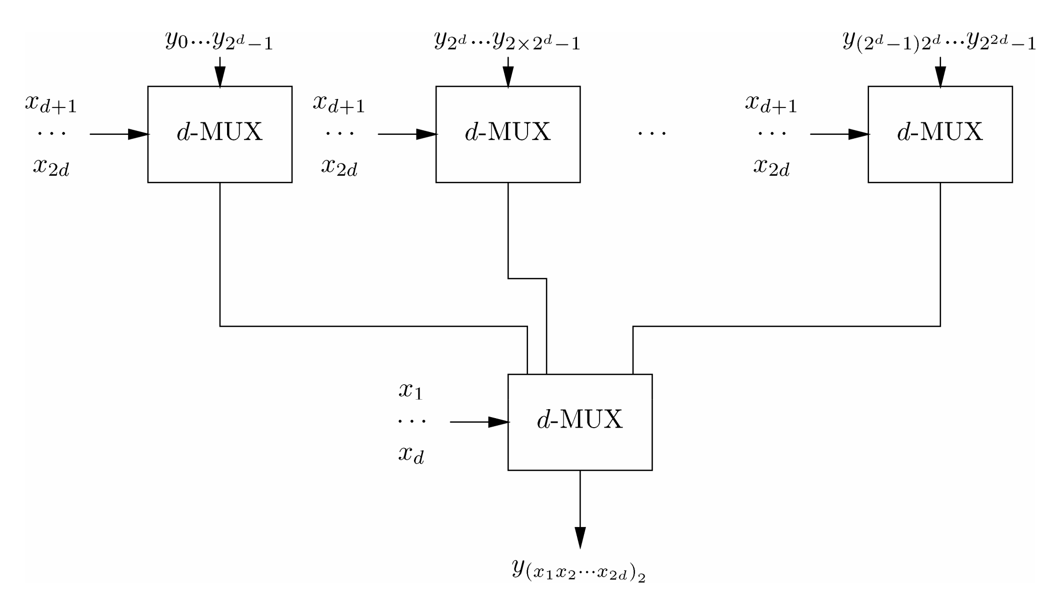


图13-24　分治多路复用器

底部*d*-MUX的输入是顶部那行*d*-MUX的输出，我们刚得出它们是 y_k 、 y_{2^d+k} 、 $y_{2\times 2^d+k}$ 、⋯、$y_{(2^d-1)2^d+k}$ 。底部的*d*-MUX是由 $x_1 \cdots x_d$ 控制的，它表示某个二进制整数*j*，也就是 $j=(x_1 \cdots x_d)_2$ 。因此底部的多路复用器会选择它的第*j*个（最左侧的输入被记作输入0）输入作为其输出。因此被选定的输入是 y_{j2^d+k} 。

可以按照如下方式证明 $j2^d+k=i$ 。请注意，用*j*乘以 2^d 会把*j*的二进制表示向左移动*d*个位置。也就是说 $j2^d=(x_1 \cdots x_d 0 \cdots 0)_2$ ，其中这串0的长度是*d*。因此， $j2^d+k$ 的二进制表示就是 $(x_1 \cdots x_d x_{d+1} \cdots x_{2d})_2$ 。这是因为*k*的二进制表示是 $(x_{d+1} \cdots x_{2d})_2$ ，而且当这个数字被加到最后是*d*个0的 $j2^d$ 时，从右起的第*d*位显然没有进位输出。现在可知 $j2^d+k=i$ ，因为它们有着相同的二进制表示。因此图13-24所示的2*d*-MUX正确地选出了 x_i ，其中 $i=(x_1 \cdots x_{2d})_2$ 。

13.7.2　分治MUX的延迟

可以通过写出适当的递推关系来计算所设计多路复用器电路的延迟。设 $D(d)$ 是*d*-MUX的延迟。观察图13-23可知，对 $d=1$ ，延迟是3。不过，要得到更紧密的边界，就要假设所有的控制输入都要经过MUX外的反相器，而且它们不能算在图13-23所示反相器的那层中。所以在确定了电路其余部分的延迟之后，要在总延迟上加1，从而把所有控制输入的反相产生的延迟计算在内。因此，我们的递推关系是从 $D(1)=2$ 开始的。

对于归纳部分，我们注意到经过图13-24所示电路的延迟是经过上方那行MUX中任何一个的延迟，加上经过最后一个MUX的延迟。因此，*D*(2d)就是*D*(d)的两倍，所以递推关系为

$$D(1)=2$$
$$D(2d)=2D(d)$$

解是很容易得出的。我们有 $D(2)=4$ ，$D(4)=8$ ，$D(8)=16$ ，而一般来说就是 $D(d)=2d$ 。当然，严格地讲，这一公式只有在*d*是2的乘方时才成立，不过同样的思路也可以用于任意数量的控制位*d*。因为我们必须加上为控制输入反相所造成的延迟1，所以该电路的总延迟就是2*d* + 1。

现在考虑简单多路复用器电路（每个数据输入对应一个AND门，其输出都提供给一个OR门）。正如之前所说的，它的延迟是3，与d无关，不过一般来说不可能这样构建电路，因为最终那个OR门的扇入是不现实的。如果坚持将扇入限制为2会怎样呢？这样一来，有着2^d路输入的最后那个OR门会被有着d层的完全二叉树替代。回想一下，这样一棵树将会有2^d个叶子节点，刚好就是正确的数量，而这棵树的延迟是d。

我们还要用由扇入为2的AND门构成的树替代这些AND门，因为一般来说这些AND门具有$d+1$路输入。回想一下，在使用具有两路输入的门时，每使用一个门就会将输入的数量减少1，所以需要d个扇入为2的门才能把$d+1$路输入减少到1路输入。如果将门安排成AND门构成的平衡二叉树，就需要$\log_2 d+1$层。在加上为控制输入反相的一层之后，就得到总延迟是$d+1+(\log_2 d+1)$。如图13-25中的表所示，虽然这与分治MUX那$2d+1$的延迟相比差别不大，但该图还是好意地对其进行了比较。

延迟		
d	分治MUX	简单MUX
1	3	3
2	5	5
4	9	8
8	17	13
16	33	22

图13-25　两种不同多路复用器设计的延迟

13.7.3　门的数量

本节中要比较简单MUX和分治MUX中门的数量。我们会看到，随着d的增加，分治MUX所含的门明显要更少。

要计算分治MUX中门的数量，可以暂时忽略反相器。我们知道，这d路控制输入各要被反相一次，所以最后再在得出的数量上加d就行了。设$G(d)$是d-MUX中（除反相器之外的）用到的门的数量。那么可以按照如下方式给出它的递推关系。

依据。依据情况是$d=1$的情况，如图13-23中的电路那样，除了反相器之外有3个门。因此$G(1)=3$。

归纳。对归纳部分来说，图13-24中的2d-MUX是用2^d+1个d-MUX构建的。

因此，递推关系就是

$$G(1)=3$$
$$G(2d)=(2^d+1)G(d)$$

正如我们在3.11节中看到过的，该递推关系的解是

$$G(d)=3(2^d-1)$$

这一递推关系的前几个值分别是$G(2)=9$，$G(4)=45$和$G(8)=765$。

现在来考虑在只使用扇入为2的门时，简单MUX使用的门的数量。和之前一样，我们会忽略为控制输入反相所需的d个反相器。最后的OR门要用一棵有2^d-1个OR门的树代替。2^d个AND门各会被一棵有d个AND门的树替代。因此，总的门数量就是$2^d(d+1)-1$。该函数要比分治MUX中门的数量多，多的幅度大约是$(d+1)/3$。图13-26比较了两种MUX中门的数量，每种情况都

不包括d个反相器。

门数量		
d	分治MUX	简单MUX
1	3	3
2	9	11
4	45	79
8	765	2303
16	196605	1114111

图13-26　两种不同多路复用器设计（不包括反相器）的门数量

有关分治的更多内容

本节的多路复用器设计所表示的这种分治算法是一种虽很少见但很强大的形式。大多数分治的例子都会把问题一分为二。这些例子包括归并排序、13.6节中设计的快速加法器，以及用来计算大量位的AND或OR的完全二叉树。在多路复用器中，是用$d+1$个更小的MUX来构建一个2d-MUX。换句话说，具有$n=2^{2d}$路数据输入的MUX是由$\sqrt{n}+1$个小MUX构建的。

13.7.4　习题

(1) 利用本节介绍的分治技巧，构建
- (a) 2-MUX
- (b) 3-MUX

(2) * 大家会如何构建那些数据输入的数量不是2的乘方的多路复用器？

(3) * 利用分治技巧设计独热码解码器（one-hot-decoder）。该电路接受d路输入x_1、x_2、…、x_d，并有2^d路输出y_0、y_1、…、y_{2^d-1}。这些输出中刚好有一个是1，具体来说就是满足$i=(x_1,x_2,\ldots,x_d)_2$的y_i。那么该电路的延迟（表示为d的函数）是多少？它要使用多少个门（表示为d的函数）？提示：有多种方法。一种电路设计方式是为前$d-1$路输入使用一个独热码解码器，并将该解码器的各输出分成基于最后一个输入x_d的两路输出。第二种方式是假设d是2的乘方，从两个独热码解码器开始，一个对应前$d/2$路输入，而另一个则对应后$d/2$路输入。然后恰当地将这些解码器的输出结合起来。

(4) * 通过为各路输出创建一个AND门，并为这些门提供恰当的输入或反相的输入，可以构建出一种独热码解码器，大家在习题(3)中设计的电路与这种显见的设计相比，延迟和门的数量各是多少？如果将大扇入的AND门用两输入的门代替，那么本题中的电路与习题(3)中设计的电路相比又是什么情况？

(5) * 多数电路（majority circuit）接受$2d-1$路输入，并只有一路输出。如果有d路或d路以上的输入是1，那么它的输出是1。设计分治多数电路。延迟和门的数量各是多少（表示为d的函数）？提示：和13.6节中的加法器一样，利用计算比我们所需更多的电路能最好地解决该问题。特别要指出的是，可以设计接受n路输入并具有$n+1$路输出y_0、y_1、…、y_n的电路。如果刚好有i个输入是1，输出y_i就是1。然后可以用习题(3)中提到的两种方式中的任意一种归纳地构建多数电路。

(6) * 有一种幼稚的多数电路设计，它是通过为每个d路输入组设置一个AND门来构建的。这种多数电路的输出是所有这些AND门的OR。与习题(5)设计的分治电路相比，这种幼稚设计的延迟和门的数量各是多少？如果用两输入门代替这种幼稚设计中的门呢？

13.8　存储单元

在结束逻辑电路这一主题之前，我们还要考虑一类非常重要的时序电路。存储单元（memory element）是由一系列的门构成的电路，它可以记住它的上一个输入，并将这个输入作为它的输出，不管这个输入已经给定多久。计算机的主存是由一些特定的位组成的，这些位可以存入值而且会保留它们的值直到另一个值被存入。

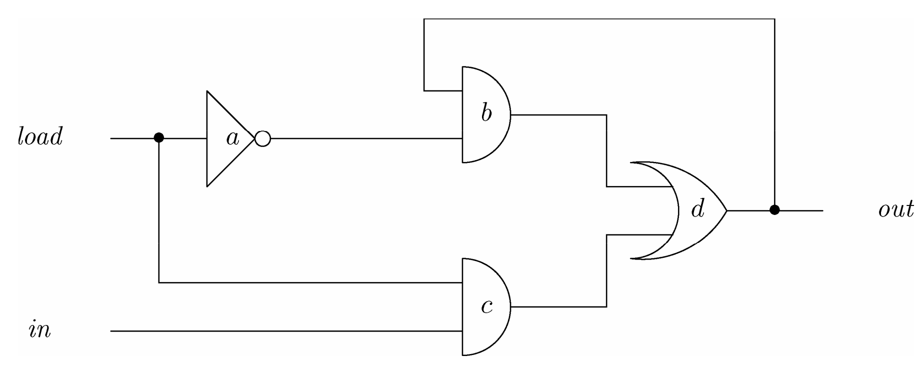


图13-27　存储单元

图13-27就展示了一个简单的存储单元。它是由称为*load*的输入控制的。一般来说，*load*的值是0。在这种情况下，反相器a的输出就是1。因为只要有一路输入是0，AND门的输出就是0，所以只要*load*是0，则AND门*c*的输出一定是0。

如果 $load = 0$ 而且门*d*的输出（也就是该电路的输出）是1，那么门*b*的两路输入都是1，这样它的输出也是1。因此，OR门*d*的输入之一是1，这样它的输出就仍然是1。另一方面，假设d的输出是0。那么AND门*b*的某一输入是0，这意味着它的输出是0。这使得*d*的两路输入都是0，所以只要 $load = 0$，*d*的输出就会保持为0。于是可以得出结论：只要 $load = 0$，电路的输出就可以保持它原来的样子。

真正的存储芯片

我们不应该认为图13-27精确地表示了典型的寄存器位，但它也不是很不靠谱。尽管它也表示了主存的位，至少原则上讲如此，但它们之间还是存在着明显的区别，而且很多与存储芯片设计有关的内容都涉及远超本书范围的电子学细节。

因为在计算机和其他类型的硬件中会用到海量的存储芯片，它们的大规模生成已经让一些存储百万位或更多位的微妙芯片设计变为现实。想知道存储芯片的致密性，可以回想一下它的面积大约是1平方厘米（10^{-4} 平方米）。如果在这样的芯片上有1600万位，那么每一位所占据的面积就等于 6×10^{-12} 平方米，或者说是2.5微米见方的一块面积，记住，1微米是 10^{-6} 米）。如果线路的最小宽度，或者说线路间的空间是0.3微米，这没有留多少空间给电路构建存储单元。更糟的是，我们不止要存储位，还要从这1600万位中选出一位接收值，或是读取这1600万位中某一位的值。这一选择电路还要占据芯片上不少空间，留给存储单元的空间就更少了。

现在考虑 $load = 1$ 时的情况。反相器*a*的输出现在是0，这样一来，AND门*b*的输出也将是0。

另一方面，AND门c的第一路输入是1，所以c的输出就与输入*in*是相同的。同样，因为OR门d的第一路输入是0，所以d的输出和c的输出是一样的，这和电路输入*in*也是相同的。因此，将*load*置为1会使电路的输出变成*in*。在把*load*变回0时，电路输出会继续在门b和d之间来回，正如前文讨论过的。

如果把“电路输入”解释为*load*为1时*in*的值，就可以说图13-27中的电路具有与存储单元相似的行为。如果*load*是0，那么可以说不管*in*的值是什么，都不存在电路输入。通过把*load*置为1，可以让该存储单元接受新值。只要*load*是0，也就是说，只要此电路没有新的输入，该单元就会保留这个值。

习题

(1) 为图13-27所示的存储单元电路画出类似图13-6的时间图。

(2) 描述一下，在如图13-27所示的存储单元中，如果一个α粒子击中反相器，并让门a的时间在一段很短的时间内与输入相同（这段时间很短，并不足以让信号传遍整个电路），该电路的行为会是怎样的。

13.9 小结

阅读本章之后，大家应该更加熟悉计算机中的电路以及逻辑是如何用来设计这种电路的。特别要说的是，本章涵盖了以下要点：

- 什么是门，以及门是如何组合起来形成电路的；
- 组合电路与时序电路间的区别；
- 如何根据逻辑表达式设计组合电路，以及如何用逻辑表达式表示组合电路的模式；
- 诸如分治法这样的算法设计技巧是如何用来设计加法器和多路复用器这样的电路的；
- 设计快速电路要考虑的一些因素；
- 简要表示了计算机是如何在它的电子电路中存储二进制位的。

13.10 参考文献

Shannon [1938]首先注意到布尔代数可以用来描述组合电路的行为。要更加全面地了解组合电路的理论与设计，参见Friedman and Menon [1975]。

Mead and Conway [1980]描述了用于构建超大规模集成电路的技术。Hennessy and Patterson [1990] 讨论了计算机体系结构以及用于组织其电路元件的技术。

Friedman, A. D. , and P. R. Menon [1975]. *Theory and Design of Switching Circuits*, Computer Science Press, New York.

Hennessy, J. L. , and D. A. Patterson [1990]. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, Calif.

Mead, C., and L. Conway [1980]. *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass.

Shannon, C. E. [1938]. “Symbolic analysis of relay and switching circuits,” *Trans. of AIEE* **57**, pp. 713–723.

第 14 章

谓词逻辑

现在要把注意力转移到一般化的命题逻辑，也就是“谓词”逻辑或者说“一阶”逻辑上。谓词是指返回布尔值的具有0个或更多变量的函数。因此，谓词可能有时为真有时为假，这取决于其参数的值。例如，我们将看到$csg(C,S,G)$这样的谓词逻辑原子操作数。其中，csg是谓词名，而C、S和G则是参数。可以将该表达式视作图8-1中数据库关系“课程–学号–成绩”的逻辑表示。只要C、S和G满足学号S的学生在课程C中得到成绩G，它就返回`TRUE`，否则返回`FALSE`。

用谓词代替命题变量作为原子操作数，提供的语言要比只涉及命题的表达式更为强大。其实，谓词逻辑的表达力足以构成很多实用编程语言的基础，比如Prolog（Programming in logic）和8.7节中我们提到过的SQL语言。谓词逻辑还应用在推理系统或“专家”系统中，比如自动化医疗诊断程序和定理证明程序。

14.1 本章主要内容

我们将在14.2节介绍谓词。谓词在正式地表示思路方面提供了比命题变量强大得多的能力。虽然存在重大差异，但谓词逻辑的设计与第12章中命题逻辑的设计是可以类比的。

- 谓词逻辑的表达式可以由使用命题逻辑运算符的谓词构建（14.3节）。
- “量词”是命题逻辑中没有类比物的谓词逻辑运算符（14.4节）。我们可以利用量词陈述某表达式对某个参数的所有值都为真，或陈述该参数至少存在一个值使得该表达式为真。
- 谓词逻辑表达式的“解释”是谓词和变量可能的含义（14.5节），它们与命题逻辑中的真值赋值是类似的。
- 谓词逻辑的重言式是指对所有解释都为真的表达式。某些谓词逻辑的重言式与命题逻辑的重言式是类似的（14.6节），而另一些则不具相似性（14.7节）。
- 谓词逻辑中的证明可以用与命题逻辑证明相类似的方式进行（14.8节和14.9节）。

14.10节要讨论谓词逻辑与计算问题解答有关的含义，我们会发现以下现象。

- 命题是重言式并不说明它在某个证明系统中是可证的。
- 特别要指出的是，哥德尔不完备性定理表明，存在某种特定形式的处理整数的谓词逻辑，在这种谓词逻辑中没有哪种证明系统可以证明每一个重言式。
- 此外，图灵定理表明，存在我们可以陈述但无法用任何计算机解决的问题。这种问题的例子之一是，某给定的C语言程序是否会在处理某些输入时进入无限循环。

14.2　谓词

谓词是对命题变量的一般化。回想一下12.10节，假设我们有3个命题：r（“天在下雨”）、u（“乔伊带着伞”）和w（“乔伊被淋湿”）。还进一步假设有3个前提，或者说我们假设为真的表达式：$r \to u$（“如果天在下雨，那么乔伊带着伞”）、$u \to \bar{w}$（“如果乔伊带伞了，那么他不会被淋湿”），以及$\bar{r} \to \bar{w}$（“如果没有下雨，乔伊不会被淋湿”）。

对乔伊为真的事情对玛丽、苏还有比尔等人也为真，因此可以把命题u看作u_{Joe}，而w就是命题w_{Joe}。如果这样看的话，就有前提

$$r \to u_{Joe},\ u_{Joe} \to \bar{w}_{Joe} \text{ 和 } \bar{r} \to \bar{w}_{Joe}$$

如果定义命题u_{Mary}表示玛丽带着她的伞，并定义w_{Mary}表示玛丽被淋湿，那么就有了一组类似的前提：

$$r \to u_{Mary},\ u_{Mary} \to \bar{w}_{Mary} \text{ 和 } \bar{r} \to \bar{w}_{Mary}$$

我们可以继续像这样，引入命题谈论所知道的所有个体X，并用新命题u_X和w_X陈述与命题r相关的前提，即

$$r \to u_X,\ u_X \to \bar{w}_X \text{ 和 } \bar{r} \to \bar{w}_X$$

现在就要讲到谓词的概念了。与无限的命题集合u_X和w_X不同的是，可以将符号u定义为接受参数X的谓词。表达式$u(X)$可以解释为在说“X带着他（她）的伞”。可能对某些X的值而言，$u(X)$为真，而对其他X的值来说，$u(X)$为假。同样，w可以是谓词，粗略地讲，$w(X)$就表示“X被淋湿”。

命题变量r也可以被当作具有0个参数的谓词。也就是说，下不下雨并不像u和w那样取决于个体X。

现在可以把前提用谓词表示成如下形式。

(1) $r \to u(X)$。（对任何个体X，如果天在下雨，那么X带着他或她的伞。）

(2) $u(X) \to \texttt{NOT}\ w(X)$。（不管你是谁，如果你带着伞，就不会被淋湿。）

(3) $\texttt{NOT}\ r \to \texttt{NOT}\ w(X)$。（如果不下雨，那么没人会被淋湿。）

14.2.1　原子公式

原子公式（atomic formula）是具有0个或更多参数的谓词。例如，$u(X)$是具有谓词u和一个参数（这里的参数是变量X）的原子公式。一般而言，参数要么是变量，要么是常量。[①]尽管原则上讲常量的值可以是任何类型的，但我们通常会假设这些值是整数、实数或字符串。

变量是那些可以接受任何常量作为其值的符号。我们不应该把“变量”与第12章“命题变量”中的“变量”弄混。事实上，命题变量等价于没有参数的谓词，而且我们会把表示原子公式的p写成具有谓词名p和0个参数的形式。

所有参数都是常量的原子公式就叫作基本原子公式（ground atomic formula）。非基本原子公式（nonground atomic formula）可以用常量或变量作为参数，但至少有一个参数一定是变量。请注意，作为没有参数的原子公式，任何命题的“所有参数都是常量”，因此是基本原子公式。

① 谓词逻辑还允许参数是单个变量或常量之外的更复杂的表达式。这对我们在本书中没有讨论到的某些用途来说是很重要的。因此，本章中我们将只会看到变量和常量作为谓词的参数。

14.2.2　常量和变量的区分

我们要使用以下约定来区分常量和变量。变量名总是以大写字母开头，常量是用以下几种方式表示的：

(1) 以小写字母开头的字符串；

(2) 12或14.3这样的数字；

(3) 带引号的字符串。

因此，如果要把课程CS101表示为常量，就可以将其写为“CS101”。[1]

像常量这样的谓词将会用以小写字母开头的字符串表示。我们不可能把谓词与常量弄混，因为常量只可能出现在原子公式的参数中，而谓词是不可能出现在那里的。

✦　示例 14.1

我们可以用谓词名csg 表示8.2节讨论过的“课程–学号–成绩”关系中所含的信息。原子公式$csg(C,S,G)$ 可以被视作在说：对变量C、S和G，学号为S的学生选修了课程C，并得到了成绩G。换句话说，当我们用常量c代替C，用s代替S，并用g代替G时，当且仅当学号为s的学生选修了课程c并取得成绩g，$csg(c,s,h)$ 的值为TRUE。

还可以通过用常量作为参数，把关系中的特定事实（即元组）表示为基本原子公式。例如，图8-1中第一个元组可以表示为$csg(\text{"CS101"},12345,\text{"A"})$，断言学号为12345的学生CS101课程的成绩是A。最后，可以在参数中混用常量与变量，因此就可能看到$csg(\text{"CS101"},S,G)$ 这样的原子公式。如果变量S和G的取值(s,g) 满足学号为s的学生选修了课程CS101并取得成绩g，则该原子公式为真，否则就为假。

14.2.3　习题

利用本节中的约定，确定以下内容是常量、变量、基本原子公式还是非基本原子公式。

(a)　CS205

(b)　cs205

(c)　205

(d)　“cs205”

(e)　$p(X, x)$

(f)　$p(3, 4, 5)$

(g)　“$p(3, 4, 5)$”

14.3　逻辑表达式

第12章中为命题逻辑使用过的概念（文字、逻辑表达式、子句等）沿用到了谓词逻辑中。在下一节中我们还会引入两种额外的运算符来构成逻辑表达式。不过，逻辑表达式构造背后的基本思路在命题逻辑和谓词逻辑中基本是相同的。

14.3.1　文字

文字要么是原子公式，要么是原子公式的否定。如果在原子公式的参数中没有变量，那么相应的文字就是基本文字（ground literal）。

---

① 常量在逻辑中通常称为“原子”。不巧的是，“原子公式”也时常被称为“原子”，因此一般会避免使用术语“原子”。

✦　示例 14.2

$p(X,a)$ 是原子公式并且是文字。它不是基本的，因为根据我们的决定，它的参数X是变量。NOT $p(X,a)$ 是文字，但它不是原子公式，也不是基本文字。表达式 $p(a,b)$ 和NOT $p(a,b)$ 都是基本文字，但只有前者是（基本）原子公式。

就像命题逻辑那样，可以用上横线代替NOT运算符。不过，当横线用在很长的表达式上时，就会容易混淆，因此与第12章相比，在本章中会更常见到NOT。

14.3.2　逻辑表达式

我们可以像12.3节中用命题变量构建表达式那样，用原子公式构建表达式。这里将继续使用第12章中讨论过的AND、OR、NOT、$\rightarrow$ 和 $\equiv$ 运算符，以及其他的逻辑连接符。而在下一节中，我们会介绍“量词”，也就是可以在谓词逻辑中用来构建表达式，但在命题逻辑中没有类比物的运算符。

就像横线是NOT的简化符号那样，可以继续用并置（没有运算符）来表示AND并用+表示OR。不过，我们并不经常使用这些简化符号，因为它们可能让谓词逻辑中较长的表达式变得难以理解。

下面的例子应该能让大家对逻辑表达式的含义有所领悟。不过，要注意到这里的讨论对其进行了非常大的简化，而我们要到14.5节才会讨论“解释”，以及它们为谓词逻辑中的逻辑表达式赋予的含义。

✦　示例 14.3

假设有谓词 csg 和 $snap$，它们分别可以解释为第8章中介绍过的“课程–学号–成绩”与“学号–姓名–地址–电话”这两个关系。并假设我们想要找到名为“C.Brown”的学生CS101课程的成绩。就可以断言以下逻辑表达式

$$\big(csg(\text{“CS101”},S,G)\ \text{AND}\ snap(S,\text{“C.Brown”},A,P)\big)\rightarrow answer(G) \qquad (14.1)$$

这里的$answer$是另一个谓词，如果G是某个名为“C.Brown”的学生CS101课程的成绩，它就适用于成绩G。

在我们“断言”某个表达式时，就说明了不管用什么值替换其变量，该表达式的值都为TRUE。粗略地讲，(14.1)这样的表达式可以按照以下方式解释。如果用常量代替各变量，则各原子公式就成了基本原子公式。通过参考“现实世界”，或是在列出某给定谓词为真的基本原子公式的关系中进行查找，可以确定一个基本原子公式是真还是假。在用0或1代替各个基本原子公式时，可以为表达式本身求值，就像第12章中为命题逻辑表达式求值那样。

在表达式(14.1)的情况中，可以取图8-1和图8-2a中的元组为真。特别要说的是，

$$csg(\text{"CS101"},12345,\text{"A"})$$

和

$$snap(12345,\text{“C.Brown”},\text{“12 Apple St.”},\text{“555}-\text{1234”})$$

为真。然后可以设

$$\begin{aligned} S &= 12345 \\ G &= \text{“A”} \\ A &= \text{“12 Apple St.”} \\ P &= \text{“555}-\text{1234”} \end{aligned}$$

这让(14.1)的左边成了1 AND 1，它的值当然是1。原则上讲，我们对谓词*answer*没有任何了解。不过，我们断言了(14.1)，这意味着不管用什么值替代其中的变量，它的值都是TRUE。因为它的左边根据上述替换得到了TRUE，所以右边不可能为FALSE。因此我们推导出了 *answer* ("A")为真。

14.3.3　其他术语

我们还会使用其他与命题逻辑相关联的术语。一般来说，当本章中讲到命题变量时，说的就是所有原子公式，其中包括不含参数的谓词（即命题变量）作为特例。例如，子句是一组由OR运算符连接的文字。同样，如果表达式是子句的AND，那么就说它是**合取范式**。如果表达式是多个项的OR，而这些项各自是文字的AND，那么这样的表达式就是**析取范式**。

14.3.4　习题

(1) 为问题“L. Van Pelt的PH100课程取得了什么成绩？”写出类似(14.1)的表达式。假设事实如图8-1和图8-2所示，其参数有什么值时能让*answer*显然为真？为展现该答案的真实性，对变量进行了怎样的替换？

(2) 设*cdh*是代表图8-2c中“课程–日子–时刻”关系的谓词，而*cr*则是对应图8-2d中“课程–教室”关系的谓词。为问题“C.Brown星期一上午9点在哪里？”（更精确地讲，是“C.Brown选修的星期一上午9点上课的课程在哪个教室上课？”）写出类似(14.1)的表达式。假设事实如图8-1和图8-2所示，其参数有什么值时能让*answer*显然为真？为展现该答案的真实性，对变量进行了怎样的替换？

(3) ** 8.7节中讨论过的各种关系代数运算可以用类似(14.1)的表达式在谓词逻辑中表示出来。例如，(14.1)本身就等价于关系代数表达式

$$\pi_{\text{成绩}}(\delta_{\text{课程="CS101" AND 姓名="C.Brown"}}(CSG \bowtie SNAP))$$

说明选择、投影、联接、并、交、差这些运算用谓词逻辑中“表达式蕴含答案”的形式表示出来是什么样子，然后将8.7节的示例中给出的各关系代数表达式转化成逻辑表达式。

14.4　量词

我们回到涉及无参数谓词*r*(“天在下雨”), 以及单参数谓词$u(X)$（“*X*带着伞”）和$w(X)$（“*X*被淋湿”）的例子。你可能希望断言“如果下雨，那么某人会淋湿”。也许会尝试

$$r \rightarrow w(\text{“乔伊”})\ \text{OR}\ w(\text{“莎莉”})\ \text{OR}\ w(\text{“苏”})\ \text{OR}\ w(\text{“山姆”})\ \text{OR}\cdots$$

但这一尝试会以失败告终，原因如下。

(1) 可以把表达式写成有限个表达式的OR，但不能把它写成无限个表达式的OR；

(2) 不知道所谈论个体的完全集。

要表示一批通过为某个变量替换所有可能的值形成的表达式的OR，就需要一种额外的方式来创建谓词逻辑表达式。这一运算符是∃，读作“存在”。我们将其用在$(\exists X)w(X)$这样的表达式中，或者粗略地将其表述为“存在某一个体*X*，满足*X*被淋湿”。一般而言，如果*E*是任何逻辑表达式，那么$(\exists X)(E)$也是逻辑表达式。[①]其大概的含义为，至少存在一个*X*的值使得*E*为真。更

① 表达式*E*两边的括号有时是必要的，有时是不必要的，这取决于该表达式的具体内容。当我们在本节稍后的内容中讨论优先级和结合性时，情况就会变得更清楚了。∃*X* 周围的括号是该符号的一部分，因此总是必需的。

精确地讲，对E中其他变量的各种取值来说，可以找出某个X的值（在所有情况中并不一定是同样的值）使得E为真。

同样，我们不能写出下面这样的无限个表达式的AND。

$$u(\text{“乔伊”})\ \text{AND}\ u(\text{“莎莉”})\ \text{AND}\ u(\text{“苏”})\ \text{AND}\ u(\text{“山姆”})\cdots$$

要构造一系列通过为某给定变量替换所有可能的值形成的表达式的AND，需要符号$\forall$（称为“对所有的”）。例如，$(\forall X)u(X)$就表示“对所有的X，X都带着伞”。一般而言，对任何逻辑表达式E，$(\forall X)(E)$意味着，对E中其他变量所有可能的取值来说，用来替换X的每个常量都能使E为真。

符号$\forall$和$\exists$就叫作*量词*。有时候也会把$\forall$叫作*全称量词*（universal quantifier），把$\exists$叫作*存在量词*（existential quantifier）。

✦ 示例 14.4

表达式$r \to (\forall X)\big(u(X)\ \text{OR}\ w(X)\big)$意味着“如果下雨，那么对所有的个体$X$，要么$X$带着伞，要么$X$被淋湿”。请注意，量词可以应用于任意表达式，而不只是前面所述例子中的原子公式。

再举个例子，可以把表达式

$$(\forall C)\Big(\big((\exists S)csg(C,X,\text{"A"})\big) \to \big((\exists T)csg(C,T,\text{"B"})\big)\Big) \tag{14.2}$$

解释为，“对所有的课程C，如果存在学号为S的学生该课程的成绩为A，那么一定存在学号为T的学生该课程的成绩为B”。不那么严谨地讲就是“如果给了A，那么也必须给B”。

第三个示例表达式是

$$\big((\forall X)\text{NOT}\ w(X)\big)\text{OR}\big((\exists Y)w(Y)\big) \tag{14.3}$$

粗略地讲就是，“要么所有个体X都不被淋湿，要么至少有一个个体Y被淋湿”。表达式(14.3)与本示例中的其他两个表达式是不同的，因为这个表达式是重言式——也就是说，不管谓词w的含义是什么，该表达式都为真。(14.3)的真实性与“雨天”的属性没有任何关系。不管使谓词w为真的值构成的集合S是什么，要么S为空（即对所有X，$w(X)$都为假），要么S不为空（也就是，存在某个Y使得$w(Y)$为真）。

14.4.1 逻辑表达式的递归定义

作为回顾，我们要给出谓词逻辑中这类逻辑表达式的递归定义。

依据。每个原子公式都是表达式。

归纳。如果E和F是逻辑表达式，那么以下表达式也是逻辑表达式。

(1) $\text{NOT}\ E$、$E\ \text{AND}\ F$、$E\ \text{OR}\ F$、$E \to F$和$E \equiv F$。粗略地讲，我们也允许使用NAND这样的其他命题逻辑运算符。

(2) 对任何变量X，$(\exists X)E$和$(\forall X)E$。原则上讲，X甚至不需要在E中出现，虽然实践中这样的表达式很难“说得通”。

14.4.2 运算符的优先级

一般而言，需要在所有用到表达式E和F的地方为其加上括号。不过，就像我们已经看到的其他代数那样，通常可以出于运算符优先级的原因删掉一些括号。这里要继续使用12.4节定义的运算符优先级，NOT（最高）、AND、OR、→和≡（最低）。不过，量词在所有运算符中有着最高的优先级。

✦ 示例 14.5

$(\exists X)p(X)$OR $q(X)$ 会被分组为

$$\big((\exists X)p(X)\big)\text{OR } q(X)$$

同样，表达式(14.3)中外层那两对括号是多余的，所以可以将其写为

$$(\forall X)\text{NOT } w(X)\text{OR } (\exists X)w(Y)$$

还可以消除(14.2)中的两对括号，并将其写为

$$(\forall C)((\exists S)csg(C,S,\text{"A"}) \rightarrow (\exists T)csg(C,T,\text{"B"}))$$

$(\forall C)$ 后整个表达式两侧的括号是必要的，这样才能把“对所有的C”应用到整个表达式上。

量词的次序

混淆量词的次序是个常见的逻辑错误，例如，有人可能误认为 $(\forall X)$ $(\exists Y)$ 与 $(\exists X)$ $(\forall Y)$ 含义相同，但它们是不同的。例如，如果粗略地把 $loves(X,Y)$ 解释为“X爱Y”，那么 $(\forall \mathrm{X})(\exists \mathrm{Y})loves(X,Y)$ 就表示“每个人都爱某个人”，也就是说，对每个个体X，至少存在一个个体Y是X所爱的。另一方面 $(\exists Y)(\forall X)loves(X,Y)$ 则表示，存在某个被每个人所爱的个体Y——这是个非常幸运的Y，如果存在这样的人的话。

请注意，量词 $(\forall X)$ 和 $(\exists X)$ 所带的括号并不是用于分组的，它们应该被看作表示量词的符号的一部分。还有，请记住量词和NOT都是一元的前缀运算符，而且唯一明智的分组方式就是从右边起为它们分组。

✦ 示例 14.6

因此表达式 $(\forall X)$ NOT $(\exists Y)p(X,Y)$ 被分组为

$$(\forall X)\Big(\text{NOT}\big((\exists Y)p(X,Y)\big)\Big)$$

并表示“对所有的X，都不存在Y使得 $p(X,Y)$ 为真”。换句话说就是，不存在使得 $p(X,Y)$ 为真的X和Y的取值对。

14.4.3　约束变量和自由变量

量词与表达式中的变量相互作用的方式是很微妙的。要解决这一问题，首先要想到C语言中局部变量和全局变量的概念。假设如图14-1所示，X被定义为C语言程序中的外部变量。假设X不是在`main`函数中声明的，那么`main`函数中对X的引用就是对外部变量的引用。另一方面，函数`f`中也声明了局部（自动控制）变量X，而函数`f`中对X的所有引用都是对该局部变量的引用。

C语言程序中对X的声明与量词 $(\forall X)$ 或 $(\exists X)$ 存在着很近的相似性。如果有表达式 $(\forall X)E$ 或 $(\exists X)E$ ，那么该量词就相当于为表达式E声明了局部的X，就像E是函数，而X被声明为该函数的局部变量那样。

在接下来的内容中，有必要用符号Q来表示任一量词。具体来说就是，用 (QX) 代表“应用于X的某个量词”，也就是 $(\forall X)$ 或 $(\exists X)$ 。

```
int X;
    ...
main()
{
    ...
    ++X;
    ...
}

void f()
{
    int X;
    ...
}
```

图14-1 局部变量和全局变量

如果E具有某个形如$(QX)F$的子表达式，那么该子表达式就像是E中声明的程序块，而E在自身对X进行了声明。在F中对X的引用就引用了由这一(QX)“声明的”X，而E中F之外的部分所使用的X则引用了X的其他声明——要么是与E相关联的量词，要么是与包含在E中但限制了所考虑的X的某个表达式相关联的量词。

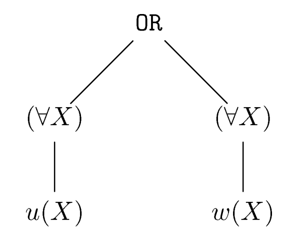


图14-2 对应$(\forall X)u(X)$ OR $(\forall X)w(X)$表达式树

✦ 示例 14.7

考虑表达式

$$(\forall X)u(X) \text{ OR } (\forall X)w(X) \tag{14.4}$$

粗略地讲，该表达式的含义是“要么每个人都带着伞，要么每个人都被淋湿”。我们可能不相信这一命题的真实性，但这里要拿它来当例子考虑。表达式(14.4)的表达式树如图14-2所示。请注意，第一个量词$(\forall X)$只在它的子孙u中使用X，而第二个量词$(\forall X)$只在它的子孙w中使用X。要区分所使用的X是在哪个量词中“声明”的，就只能从该X向上追溯，直到遇到量词(QX)为止。因此这里所使用的两个X引用了不同的“声明”，而且它们之间没有任何关系。

要注意可以为(14.4)中X的两个“声明”使用不同变量，将其写作$(\forall X)u(X)$ OR $(\forall Y)w(Y)$。一般来说，总是可以为谓词逻辑表达式的变量重命名，从而使同一变量不会出现在两个量词中。这种情况与C语言这样的编程语言是类似的，我们在编程语言中会为程序中的变量重命名，这样相同的变量名就不会使用在两个声明中。例如，在图14-1中，可以把函数`f`中变量名`X`的所有实例都变为任何新变量名`Y`。

✦ 示例 14.8

再举个例子，考虑表达式

$$(\forall X)\big(u(X) \text{ OR } (\exists X)w(X)\big)$$

粗略地讲，其含义是“对各个体，要么该个体带着伞，要么存在某一（可能是另一）个体被淋湿”。该表达式的表达式树如图14-3所示。请注意，w中使用的X指的是X限定了私密性的“声明”，也就是存在量词。换句话说，如果从 $w(X)$ 沿着树向上行进，那么在遇到全称量词之前会遇到存在量词。不过，u中所使用的X就不在该存在量词的“范围”内。如果从 $w(X)$ 上行，首先会遇到全称量词。可以把该表达式写为

$$(\forall X)\big(u(X)\ \mathrm{OR}\ (\exists Y)w(Y)\big)$$

这样就没有哪个变量会出现在两个量词中了。

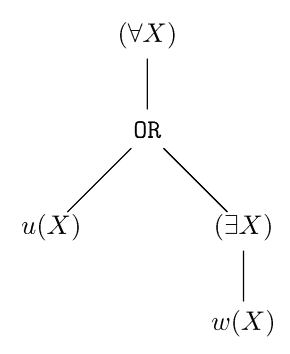


图14-3　对应 $(\forall X)\big(u(X)\ \mathrm{OR}\ (\exists X)w(X)\big)$ 的表达式树

如果在逻辑表达式E的表达式树中，涉及某个变量X的量词是该X的最低祖先，就可以说该变量X是受量词 $Q(X)$ 约束的。如果某个X不受任何量词约束，那么该X就是自由变量。因此量词就像是以该量词为根节点的子树T局部的“声明”。这些量词会应用到T中除了以具有同样变量的另一个量词为根节点的子树之外的各个节点。而自由变量就像是全局变量之于某一函数那样，它们的“声明”是在所考虑的表达式之外进行的。

✦ 示例 14.9

考虑表达式

$$u(X)\ \mathrm{OR}\ (\exists X)w(X)$$

也就是说，“要么X带着伞，要么有某个人会被淋湿”。相应的表达式树如图14-4所示。正如之前的例子中那样，这里出现的两个X指的是不同个体。w中出现的X是受到存在量词约束的。不过，在u中出现的X之上没有对应X的量词，因此这次出现的X在给定的表达式中是自由变量。这个例子说明，在某表达式中同一变量可能同时作为自由变量和作为约束变量出现，所以在某些情况下，我们会说“作为约束变量出现”而不是直接说“约束变量”。示例14.7和14.8中的表达式表明，出现在不同位置的相同变量，也可能分别受到出现在不同位置的相同量词的约束。

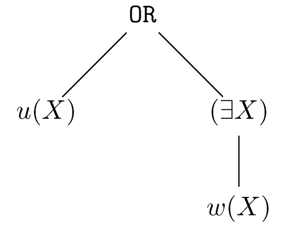


图14-4　对应 $u(X)\ \mathrm{OR}\ (\exists X)w(X)$ 的表达式树

14.4.4 习题

(1) 从以下表达式中删除多余的括号。

(a) $(\forall X)\Big((\exists Y)\Big(\texttt{NOT}\big(p(X)\texttt{OR}\big(p(Y)\texttt{AND}\ q(X)\big)\big)\Big)\Big)$

(b) $(\exists X)\Big(\big(\texttt{NOT}\ p(X)\big)\texttt{AND}\big((\exists Y)\big(p(Y)\big)\texttt{OR}\big(\exists X\big)\big(q(X,Z)\big)\big)\Big)$

(2) 为习题(1)中的表达式画出表达式树。如果出现的变量是受量词约束的，则指出它是受哪个量词约束的。

(3) 重写习题(1)中的表达式(b)，使得其中的量词不含相同的变量。

(4) * 在前文附注栏“量词的次序”中，我们谈论了量词 $loves(X,Y)$ ，并为其给出了预料之中的粗略解释。不过，正如我们将在14.5节中看到的，谓词没有具体的解释，而且也可以拿 $loves$ 来谈论整数而非个人，并为 $loves(X,Y)$ 给出 $Y = X + 1$ 这样的粗略解释。在这种解释下，比较 $(\forall X)(\exists Y)loves(X,Y)$ 和 $(\exists Y)(\forall X)loves(X,Y)$ 的含义。它们的粗略解释各是什么？如果可能的话，大家会相信哪个？

(5) * 利用之前例子中的谓词csg，写出断言以下内容的表达式。

(a) C.Brown是个A等生（即他所有课程的成绩都是A）。

(b) C.Brown不是A等生。

(6) * 设计文法，描述合法的谓词逻辑表达式。大家可以使用常量和变量这样具有象征性的终结符，而且不需要考虑重复括号的问题。

14.5 解释

直到现在，我们对谓词逻辑表达式有何“含义”，或者说是对如何为表达式赋予含义的了解还是相当模糊。这里要通过先回顾命题逻辑表达式E的“含义”来阐释这一主题。命题逻辑表达式的含义是接受“真值赋值”（为E中的命题变量指定真值0和1的情况）作为参数，并产生0或1作为结果的函数。根据给定的真值赋值，用0或1替代表达式E中的各原子操作数，并求出E的值，从而确定结果。换句话说，逻辑表达式E的含义就是为各组真值赋值给出相应E值（0或1）的真值表。

而真值赋值是接受命题变量作为参数，并为各参数返回0或1的函数。换句话说，可以把真值赋值视为给各命题变量给定某一真值（0或1）的表格。图14-5展示了这两种函数的角色。

$$p \xrightarrow{\text{真值赋值}} 0\text{或}1$$

(a) 真值赋值是从谓词到真值的函数

$$\text{真值赋值} \xrightarrow{\text{含义}} 0\text{或}1$$

(b) 表达式的含义是从真值赋值到真值的函数

图14-5 命题逻辑中表达式的含义

在谓词逻辑中，为谓词指定常数0或1（TRUE或FALSE）是不够的，除非谓词不含参数，而这种情况下它们从本质上讲就是命题变量。不过，为谓词赋的值本身是歌函数，它接受谓词参数的值作为输入，并产生0或1作为输出。

更加精确地讲，首先必须从变量可能的取值中选取一些值构成非空的定义域D。这一定义域可以是任何内容：整数、实数，或由没有特殊名称或意义的值构成的某个集合。不过，假设定义域含有出现在表达式本身中的所有常量。

现在，设p是具有k个参数的谓词。那么谓词p的解释就是接受定义域元素到p中k个参数的赋值作为输入，并返回0或1（TRUE或FALSE）的函数。或者说，可以把p的解释看作具有k列的关系。对让p在该解释中为真的各参数赋值来说，在该关系中都存在相应的元组。[①]

现在可以把表达式E的解释定义为：

(1) 非空的定义域D，含有E中出现的任何常量，

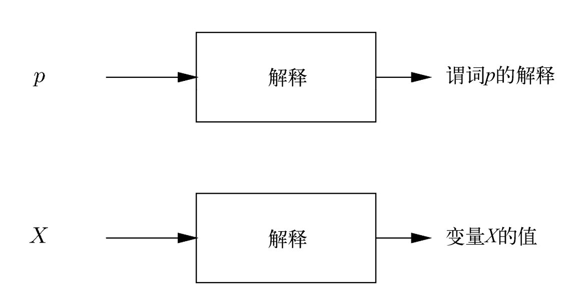
(2) 对E中出现的各谓词p的解释，以及

(3) D中对应表达式E各自由变量的值（如果存在自由变量的话）。

解释与“对谓词的解释”分别如图14-6a和图14-6b所示。请注意，解释在谓词逻辑中的角色就相当于真值赋值在命题逻辑中的作用。



(a) 谓词的解释为参数值元组指定了真值



(b) 解释为各谓词指定了谓词解释，并为各变量指定了值（类似真值赋值）



(c) 表达式的含义为各解释指定真值（类似真值表）

图14-6　谓词逻辑中表达式的含义

✦ 示例 14.10

考虑以下谓词逻辑表达式：

$$p(X,Y) \to (\exists Z)\big(p(X,Z) \text{ AND } p(Z,Y)\big) \tag{14.5}$$

谓词p一种可能的解释（我们将称其为解释I_1）如下所述。

(1) 定义域D是实数集。

(2) 只要$U<V$，$p(U,V)$就为真。也就是说，p的解释是有序对(U,V)的无限集构成的关系，其中满足U和V都是实数而且U小于V。

① 与第8章中谈论的关系不同，作为谓词解释的关系可能具有无限多的元组。

那么(14.5)就表示，对任何实数X和Y，如果$X<Y$，则存在某个Z严格位于X和Y之间，也就是说，有$X<Z<Y$。对解释I_1而言，(14.5)总是为真。如果$X<Y$，就可以选择$Z=(X+Y)/2$，也就是X和Y的平均数，然后就能确定$X<Z$且$Z<Y$。如果$X\geqslant Y$，那么该蕴涵式的左边为假，则(14.5)显然为真。

我们可以根据谓词p的解释I_1，通过选择任何实数作为自由变量X和Y，为(14.5)构建无数的解释。根据我们刚才所说的，这些对应(14.5)的解释都能使(14.5)为真。

谓词p第二种可能的解释I_2如下：

(1) D是整数集；

(2) 当且仅当$U<V$时，$p(U,V)$为真。

现在，我们可以声明(14.5)为真，除非$Y=X+1$。因为如果Y比X大2或者更多，那么Z就可以被选为$X+1$。这样就是满足$X<Z<Y$的情况。如果$X\leqslant Y$，那么$p(X,Y)$为假，则(14.5)还是为真。不过，如果$Y=X+1$，那么$p(X,Y)$为真，但不存在严格处于X和Y之间的整数Z。因此这种情况下对每个整数Z而言，$p(X,Z)$和$p(Z,Y)$都为假，则蕴涵式的右边，即$(\exists Z)\big(p(X,Z)\,\texttt{AND}\,p(Z,Y)\big)$不为真。

通过为自由变量X和Y赋整数值，可以将I_2扩展为表达式(14.5)的解释。以上分析说明了，除了$Y=X+1$情况下外，(14.5)对任何这样的解释都为真。

对p的第三种解释I_3是抽象的，不像之前的解释I_1和I_2那样具有常见的数学含义。

(1) D是三个符号a、b、c的集合。

(2) 如果UV是aa、ab、ba、bc、cb、cc其中之一，则$p(U,V)$为真，若UV为ac、bb和ca，则$p(U,V)$为假。那么刚好有(14.5)对9对XY都为真。在每种情况下，要么$p(X,Y)$为假，要么存在Z使得(14.5)的右边为真。图14-7列举了这9种情况。通过为自由变量X和Y指定由a、b、c构成的任意赋值组合，我们有9种方式可以把I_3扩展为(14.5)的解释。

X	Y	为何为真
a	a	$Z=a$ 或b
a	b	$Z=a$
a	c	$p(a,c)$ 为假
b	a	$Z=a$
b	b	$p(c,a)$ 为假
b	c	$Z=c$
c	a	$p(b,b)$ 为假
c	b	$Z=c$
c	c	$Z=b$ 或c

图14-7　使用解释I_3的情况下(14.5)的值

14.5.1　表达式的含义

回想一下，命题逻辑中表达式的含义就是从真值赋值到真值0和1的函数，如图14-5b所示。也就是说，真值赋值陈述了与表达式原子操作数的值有关的所有信息，然后为该表达式求值得

到0或1。同样，在谓词逻辑中，表达式的含义是接受解释（我们需要利用该解释为原子操作数求值），并返回0或1的函数。表达式含义的这一概念如图14.6c所示。

✦ 示例 14.11

考虑示例14.10中的表达式(14.5)。(14.5)中的自由变量是X和Y。如果给定的是示例14.10中对p的解释I_1（p是针对实数的<），而且给定了值$X = 3.14$和$Y = 3.5$，那么(14.5)的值就是1。其实，正如我们在示例14.10中讨论过的，有着对p的解释I_1，任何X和Y的值都使该表达式的值是1。同样的结论也适用于对p的解释I_3，从定义域$\{a，b，c\}$中选取的任何X和Y的值都会让(14.5)的值为1。

另一方面，如果给定了解释I_2（p是针对整数的<），以及值X=3和Y=4，那么就像我们在示例14.10中讨论过的，(14.5)的值是0。如果有解释I_2，而且自由变量的值分别是X=3和 Y=5，那么(14.5)的值是1。

要完成对表达式“含义”的定义，必须正式地定义如何把原子操作数的指针转化成整个表达式的真值。之前我们已经利用直觉，根据的是对命题逻辑的逻辑连接符作用方式的理解，以及考虑量词的直觉。给定某解释I以及定义域D，表达式的值的正式定义就是对给定的表达式E的表达式树进行的结构归纳。

依据。如果表达式树是单个叶子节点，那么E是原子公式$p(X_1,\cdots,X_k)$。这些X_i全都要么是常量、要么是表达式E的自由变量。解释I为各变量给定了值，这样一来就拥有了p所有参数的值。同样，I表明了在以这些值作为参数的情况下p是真还是假。而该真值就是表达式E的值。

归纳。现在，必须假设给定的表达式E对应的表达式树根节点位置是运算符。这存在若干种情况，具体取决于E的根节点位置是什么运算符。

首先，考虑一下E形如E_1 AND E_2的情况，也就是说，根节点处的运算符是AND。归纳假设可以应用于子表达式E_1和E_2。因此可以在解释I之下为E_1求值。[①]同样，可以在解释I之下为E_2求值。如果求出的值都是1，那么E的值就是1，否则E的值是0。

像OR或NOT这样的其他逻辑运算符的归纳也是如法炮制。对OR而言，我们会为两个子表达式求值，并且只要有任何一个子表达式得出值1，就为表达式得出值1。而对NOT来说，我们会为那一个子表达式求值，并得出该表达式的值的否定，而对其他命题逻辑运算符来讲，处理方法也都是一样的。

现在假设E形如$(\exists X)E_1$。根节点运算符就是该存在量词，而且我们可以将归纳假设应用于子表达式E_1。E_1中的谓词都出现在E中，而E_1中的自由变量都是E的自由变量，可能还要加上X。[②]因此，我们可以为定义域D中的各个值v构建对应E_1的解释I，以及我们称之为解释J_v的对变量X的赋值v。对各个值v，我们会问，在解释J_v之下E_1是否为真。如果至少存在一个这样的值v，那么我们说$E = (\exists X)E_1$为真，否则就说E为假。

最后，假设E形如$(\forall X)E_1$。归纳假设还是适用于E_1。现在要问，对定义域D中的每个值v，在解释J_v之下E_1是否为真。如果是，就说E的值是1，如果不是，E的值就是0。

---

① 严格地讲，要从I中除去那些只出现在E中但没有出现在E_1中的对应谓词p的解释。还有，必须放弃那些出现在E中但没有出现在E_1中的自由变量的值。不过，如果解释中包含进没有用到的额外信息，是不存在任何概念困难的。

② 技术上讲，即使对E_1应用了涉及X的量词，E_1还是可能不含任何作为自由变量的X。在这种情况下，量词可能也不存在，但我们没有阻止它出现。

✦ 示例 14.12

这里在给定对p的解释I_2，以及对应自由变量X和Y的值分别是3和7的情况下，要为表达式(14.5)求值。对应(14.5)的表达式树如图14-8所示。我们看到，根节点处的运算符是→。之前并未明确介绍过这种情况，不过原则应该是很清楚的。整个表达式可以写成$E_1 \to E_2$，其中E_1是$p(X,Y)$，而E_2是$(\exists Z)\big(p(X,Z)\text{AND}p\ (Z,Y)\big)$。因为→的含义，所以除了$E_1$为真而且$E_2$为假的情况，整个表达式(14.5)都为真。

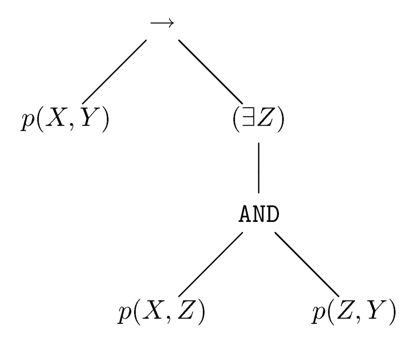


图14-8 对应(14.5)的表达式树

E_1，也就是$p(X,Y)$，是很容易求值的。因为$X=3$，$Y=7$，而且当且仅当$X<Y$时$p(X,Y)$为真，所以可以得出E_1为真。为E_2求值则更为困难。我们必须为Z考虑所有可能的值v，以了解是否至少存在一个值使$p(X,Z)$ AND $p(Z,Y)$为真。例如，如果尝试$Z=0$，那么$p(Z,Y)$为真，但$p(X,Z)$为假，因为$X=3$是不小于Z的。

能否计算表达式的值？

大家可能会怀疑在E是$(\exists X)E_1$或$(\forall X)E_1$的情况下，我们对表达式E值为1的定义。如果定义域D是无限的，那么我们已经提出的在各解释J_v之下为E_1求值的测试就不需要对应其执行的算法。从本质上讲，要求我们为存在量词执行以下函数

```
for (each v in D)
    if (E1 is true under interpretation Jv)
        return TRUE;
return FALSE;
```

并为全称量词执行如下函数

```
for (each v in D)
    if (E1 is false under interpretation Jv)
        return FALSE;
return TRUE;
```

尽管这些程序的目的很明确，但它们都不是算法，因为若定义域D是无限的，则要进行无数次循环。不过，虽然可能没法分辨E是真还是假，但我们还是给出了E何时为真的正确定义，也就是说，我们为量词∀和∃赋予了预期的含义。在很多实际且实用的情形中，我们将能够分清E是真还是假。在另一些情况中，我们会看到E是真还是假都是没关系的，例如涉及将表达式变形为等价形式的情况。可以在不知道是否存在令E_1这样的子表达式为真的值v的情况下，根据两个表达式的值的定义来推理它们是等价的。

如果考虑这种情况，就会看到，要使 $p(X,Z)$ AND $p(Z,Y)$ 为真，就需要v的值满足$3<v$（从而让 $p(X,Z)$ 为真），并满足$v<7$（从而使 $p(Z,Y)$ 为真）。例如，$v=4$就能使 $p(X,Z)$ AND $p(Z,Y)$ 为真，并因此证明了 E_2，或者说证明了 $(\exists Z)\big(p(X,Z) \text{ AND } p(Z,Y)\big)$ 对给定的解释而言为真。

现在可知 E_1 和 E_2 都为真。因为当 E_1 和 E_2 都为真时 $E_1 \to E_2$ 为真，所以可以得出结论，在谓词p具有解释 I_2，而且 $X=3$，$Y=3$ 的情况下，表达式(14.5)的值是1。

14.5.2　习题

(1) 分别为以下各表达式给出一种使其为真的解释以及一种使其为假的解释。

(a) $(\forall X)(\exists Y)\big(loves(X,Y)\big)$

(b) $p(X) \to \text{NOT } p(X)$

(c) $(\exists X)p(X) \to (\forall X)p(X)$

(d) $\big(p(X,Y) \text{AND } p(Y,Z)\big) \to p(X,Z)$

(2) 解释一下，为什么每种解释都能使表达式 $p(X) \to p(X)$ 为真。

14.6　重言式

回想一下，在命题逻辑中，如果对每种真值赋值而言，表达式的值都是1，就说该表达式是重言式。同样的概念在谓词逻辑中也是成立的。如果对E的每种解释，E的值都是1，则说表达式E是重言式。

✦　示例 14.13

就像在命题逻辑中那样，“随机的”谓词逻辑表达式很少是重言式。例如，我们在示例14.10中研究过的表达式(14.5)，或者说

$$p(X,Y) \to (\exists Z)\big(p(X,Z) \text{AND } p(Z,Y)\big)$$

在某些针对谓词p的解释之下总为真，但是存在像示例14.10中的 I_2 这样的解释：p是针对整数的<，让该表达式不总是为真，比如，对 $X=1$ 和 $Y=2$，该表达式为假。因此，该表达式不是重言式。

表达式

$$q(X) \text{ OR NOT } q(X)$$

就是个重言式。这里，不管为谓词q使用什么解释，或者为自由变量X赋什么值，都是没关系的。如果所选择的解释使得$q(X)$为真，那么该表达式为真。

14.6.1　替换原则

命题逻辑重言式是谓词逻辑重言式的丰富来源。我们在12.7节中介绍过的替换原则表明，可以取任意命题逻辑重言式，对其中的命题变量进行任何替换，得到的结果仍然是重言式。如果允许用谓词逻辑表达式替换命题变量，那么该原则仍然成立。例如，示例14.13中提到过的重言式$q(X)$OR NOT $q(X)$，就是用表达式$q(X)$替换了重言式p OR NOT p中的命题变量p。

用谓词逻辑表达式替换命题变量时替换原则成立的原因，与用命题表达式替换命题变量时该原则成立的原因基本相同。在用$q(X)$这样的表达式替代表达式中出现的某一命题变量p时，我们知道，对任何解释来说，所替换的表达式不管出现在何处都有着相同的值。因为原有的（我

们要对其进行替换的）命题逻辑表达式是重言式，所以在将其中某个命题变量全部替换为0，或者全部替换为1时，该表达式总是为真。

例如，在表达式$q(X)$ OR NOT $q(X)$中，不管q的解释是什么或X的值是什么，$q(X)$要么为真，要么为假。因此，这个表达式要么变成1 OR NOT 1，要么变成0 OR NOT 0，而这两个式子的值都是1。

14.6.2 表达式的等价

和命题逻辑中一样，如果两个谓词逻辑表达式E和F满足$E \equiv F$是重言式，就可以定义它们是等价的。在讲到谓词逻辑表达式等价时，12.7节中介绍过的“以相等换相等原则”继续成立。也就是说，如果E_1等价于E_2，那么可以用E_2代替任一表达式F_1中的E_1，得到的表达式F_2将是等价的，也就是说$F_1 \equiv F_2$。

✦ 示例 14.14

AND运算的交换律是$(p \text{ AND } q) \equiv (q \text{ AND } p)$。现在，如果有$\big(p(X) \text{AND } q(Y,Z)\big) \text{OR } q(X,Y)$这样的表达式，那么可以用$q(Y,Z) \text{ AND } p(X)$替换$p(X) \text{ AND } q(Y,Z)$，产生另一个表达式

$$\big(q(Y,Z) \text{ AND } p(X)\big) \text{ OR } q(Y,Z)$$

并知道

$$\big((p(X)\text{AND } q(Y,Z))\text{OR } q(X,Y)\big) \equiv \big((q(Y,Z)\text{AND } p(X)\big)\text{OR } q(X,Y)\big)$$

还有一些更微妙的等价谓词逻辑表达式。通常，我们会认为等价表达式有着相同的自由变量和谓词，不过有些情况下自由变量和（或）谓词可以是不同的。例如，表达式

$$\big(p(X)\text{OR NOT } p(X)\big) \equiv \big(q(Y)\text{OR NOT } q(Y)\big)$$

就是重言式，因为$\equiv$两边的表达式如我们在示例14.13中论证过的都是重言式。因此，在表达式$p(X)\text{OR NOT } p(X)\text{OR } q(X)$中可以用$q(Y)\text{OR NOT } q(Y)$替换$p(X)\text{OR NOT } p(X)$，从而得到

$$\big(p(X)\text{OR NOT } p(X)\text{OR } q(X)\big) \equiv \big(q(Y)\text{OR NOT } q(Y)\text{OR } q(X)\big)$$

因为$\equiv$的左边是重言式，所以还可以得出

$$q(Y)\text{OR NOT } q(Y)\text{OR } q(X)$$

是重言式的结论。

14.6.3 习题

解释以下各表达式为何是重言式。也就是说，我们为命题逻辑重言式替换了什么样的谓词逻辑表达式？

(a) $\big(p(X)\text{OR } q(Y)\big) \equiv \big(q(Y)\text{OR } p(X)\big)$

(b) $\big(p(X,Y)\text{AND } p(X,Y)\big) \equiv p(X,Y)$

(c) $\big(p(X) \to \text{FALSE}\big) \equiv \text{NOT } p(X)$

14.7 涉及量词的重言式

涉及量词的谓词逻辑重言式在命题逻辑中没有直接的参照物。本节要探究这些重言式，并展示如何利用它们处理表达式。而主要成果就是可以将任何表达式转换成量词全在开头位置的等价表达式。

14.7.1　变量的重命名

在C语言中，可以改变局部变量的名称，只要对所有用到该局部变量的地方进行统一修改即可。类似地，也可以改变量词中用到的变量，只要对所有出现受到该量词约束的这一变量的地方进行修改。还有，就像在C语言中那样，一定要谨慎选择新的变量名，因为如果选择的名称已经在所考虑函数之外定义，那么就可能改变程序的含义，就会造成严重的错误。

记住这种重命名，我们可以考虑以下类型的等价，以及在何条件下它是重言式。

$$(QX)E \equiv (QY)E' \tag{14.6}$$

其中 E' 是把E中所有受到这里明确给出的量词 (QX) 约束的X替换为Y后得到的。我们说(14.6)是重言式，只要E中没有出现自由变量Y。要知道原因，可以考虑任一对应 $(QX)E$ 的解释I。或者等价地，对应 $(QY)E'$，因为两个量词化表达式的自由变量和谓词是相同的。如果通过为X给定值v扩展过的I使得E为真，那么I和对应Y的值v也能让 E' 为真。相反，如果通过为X给定值v扩展的I使E为假，那么扩展的I和对应Y的v也使 E' 为假。

如果量词Q是$\exists$，那么假设存在对应X的值v使得E为真，就存在某个对应Y的值v，使得 E' 为真，相反，假设存在X的值v使得E为假，就存在对应Y的值v使得 E' 为假。如果Q是$\forall$，那么当且仅当所有的Y值使 E' 为真时，所有的X值使E为假。因此，对任一量词而言，在给定的任一解释I之下，当且仅当 $(QY)E'$ 在相同解释下为真时，$(QX)E$ 才为真，这就证明了如下表达式是重言式。

$$(QX)E \equiv (QY)E'$$

✦ 示例 14.15

考虑刚好是重言式的表达式

$$\big((\exists X)p(X,Y)\big)\texttt{OR NOT}\big((\exists X)p(X,Y)\big) \tag{14.7}$$

我们要展示如何为这两个X中的一个重命名，以形成两个量词中使用不同变量的另一个重言式。

如果设表达式(14.6)中的E是 $p(X,Y)$，并且选择变量Z扮演(14.6)中Y的角色，就有重言式 $\big((\exists X)p(X,Y)\big)\equiv\big((\exists Z)p(Z,Y)\big)$。也就是说，如果要构造表达式 E'，就要用Z替换 $E=p(X,Y)$ 中的X，从而得到 $p(Z,Y)$。因此，可以“以相等换相等”，把(14.7)中的第一个 $(\exists X)p(X,Y)$ 替换为 $(\exists Z)p(Z,Y)$，以得出表达式

$$\big((\exists Z)p(Z,Y)\big)\texttt{OR NOT}\big((\exists X)p(X,Y)\big)$$

该表达式等价于(14.7)，因此也是重言式。

请注意，也可以用Z替换(14.7)第二半中的X，是否这样做都是无关紧要的，因为这两个量词定义了没有关系的不同变量，只不过在(14.7)中它们都被命名为X。不过，我们应该明白，任何一个 $\exists X$ 都不能用 $\exists Y$ 替代，因为在出现的两个子表达式 $p(X,Y)$ 中都是自由变量。

也就是说，$\big((\exists X)p(X,Y)\big)\equiv c$ 不是重言式(14.6)的实例，因为Y是 $p(X,Y)$ 中的自由变量。要知道这为什么不是重言式，可以设把p解释为针对整数的<。那么对自由变量Y的任何值，比方说 $Y=10$，表达式 $(\exists X)p(X,Y)$ 都为真，因为我们可以设 $X=9$。不过该等价的右边——$(\exists Y)p(X,Y)$——为假，因为没有哪个整数严格小于自身。

同样，也不能用 $(\exists Y)p(X,Y)$ 替换(14.7)中 $(\exists X)p(X,Y)$ 的第一个实例。因为得到的表达式

$$\big((\exists Y)p(Y,Y)\big)\texttt{OR NOT}\big((\exists X)p(X,Y)\big) \tag{14.8}$$

也不可能是重言式。这里还是设p的解释为针对整数的<，并设自由变量Y的值是10。请注意，在(14.8)中，出现在$p(X,Y)$中的两个Y都是受到量词$(\exists Y)$约束的，只有出现在$p(X,Y)$中的最后一个Y是自由的。那么$(\exists Y)p(Y,Y)$对这一解释而言为假，因为没有Y的值比它自身小。另一方面，当$Y=10$（或其他任何整数）时，$(\exists X)p(X,Y)$为真，所以$\text{NOT}\big((\exists X)p(X,Y)\big)$为假。这样一来，表达式(14.8)对这一解释而言为假。

让量词化的变量唯一

(14.6)式有个有意思的结果，就是我们总是可以把任何谓词逻辑表达式E转换成等价表达式，使其没有两个量词使用同一变量，而且没有量词使用在E中也作为自由变量的变量。这样的表达式称为修正表达式（rectified expression）。

在证明过程中，我们可能从重言式$E \equiv E$开始，然后利用(14.6)，以E中未使用过的新变量，依次为右边的E中各量词化的变量重命名。得到的结果是表达式$E \equiv E'$，其中E'中所有的量词(QX)都涉及不同的X，而这些X也不是E或E'中的自由变量。根据命题逻辑中等价的传递性，$E \equiv E'$是重言式，也就是说E和E'是等价的表达式。

14.7.2 自由变量的全称量词化

只有在其自由变量全称量词化的相同表达式为重言式时，带有自由变量的表达式才可能是重言式。严格来讲，对所有重言式T和变量X而言，$(\forall X)T$也是重言式。技术上讲，出现在T中的X是否自由都是没关系的。

要知道$(\forall X)T$为什么是重言式，设Y_1、…、Y_k是T的自由变量，X可能是其中一员，也可能不是。首先，假设$X=Y_1$。我们需要证明，对所有的解释I，$(\forall X)T$为真。等价地讲，也就是需要证明，对I的定义域中的每个值v，通过为X给定值v而由I形成的解释J_v使T为真。但T是重言式，所以每种解释都会使其为真。

如果X是T的其他自由变量Y_i中的某一个，那么论证$(\forall X)T$为重言式的过程本质上讲是相同的。如果X不是Y_i中的一员，那么它的值不会对T的真假产生影响。因此T对所有的X都为真，就因为T是重言式。

14.7.3 闭表达式

一个有意思的结果是，对重言式来说，可以假设不存在自由变量。我们可以利用之前的变形，一次全称量词化一个自由变量。不含自由变量的表达式就叫作闭（closed）表达式。

✦ 示例 14.16

我们知道$p(X,Y)\text{OR NOT}p\ (X,Y)$是重言式。可以为自由变量X和Y加上全称量词，得到重言式

$$(\forall X)(\forall Y)(p(X,Y)\text{OR NOT}p\ (X,Y))$$

14.7.4 把量词移过NOT

存在德摩根律的无限版本，让我们可以用$\exists$替代$\forall$，反之亦然，就像“普通的”德摩根律

允许我们在移过NOT的时候，在AND和OR之间切换一样。假设有像

$$\text{NOT}\big((\forall X)p(X)\big)$$

这样的表达式。如果定义域值的数量是有限的，比方说是v_1、…、v_n，那么可以把该表达式看作$\text{NOT}\big(p(v_1)\text{AND } p(v_2)\text{AND}\cdots\text{AND } p(v_n)\big)$。然后，可以应用德摩根律把该表达式重新写为$\text{NOT } p(v_1)\text{OR NOT } p(v_2)\text{OR}\cdots\text{OR NOT } p(v_n)$。在有限定义域的假设之下，这一表达式就等同于$(\exists X)\big(\text{NOT } p(X)\big)$，也就是说，对某个$X$的值，$p(X)$为假。

事实上，这一变形并不取决于定义域的有限性，它对每种可能的解释来说都是成立的。也就是说，下面的等价对任何表达式E来说都是重言式。

$$\Big(\text{NOT}\big((\forall X)E\big)\Big) \equiv \big((\exists X)(\text{NOT } E)\big) \tag{14.9}$$

粗略地讲，(14.9)表示，刚好在存在某个X的值令E为假时，E对所有的X都不为真。

还有一个类似的重言式让我们可以把NOT压入存在量词。

$$\Big(\text{NOT}\big((\exists X)E\big)\Big) \equiv \big((\forall X)(\text{NOT } E)\big) \tag{14.10}$$

粗略地讲就是，刚好当E对所有X来说都为假时，不存在X使E为真。

✦ 示例 14.17

考虑我们根据命题逻辑重言式$p \text{ OR NOT } p$，利用替换原则得到的重言式

$$(\forall X)p(X)\text{OR NOT}\big((\forall X)p(X)\big) \tag{14.11}$$

可以令(14.9)中的E=$p(X)$，用$(\exists X)\big(\text{NOT } p(X)\big)$替换(14.11)中的$\text{NOT}\big((\forall X)p(X)\big)$，得到重言式

$$(\forall X)p(X)\text{OR}(\exists X)\big(\text{NOT } p(X)\big)$$

也就是说，要么$p(X)$对所有的X都为真，要么存在某个X令$p(X)$为假。

14.7.5 把量词移过AND和OR

在从左向右应用法则(14.9)和(14.10)时，可以把量词移到否定之外，并在这样做的过程中“反转”量词，也就是用$\forall$代替$\exists$，用$\exists$代替$\forall$。同样，可以把量词移到AND或OR之外，不过一定要小心，不能改变其中出现的任何变量的约束情况。还有，我们在移过AND或OR时不会反转量词。这些法则的表达式为

$$\big(E\text{AND}(QX)F\big) \equiv (QX)(E \text{ AND } F) \tag{14.12}$$

$$\big(E \text{ OR}(QX)F\big) \equiv (QX)(E \text{ OR } F) \tag{14.13}$$

其中E和F是任何表达式，而Q是任一量词。不过，我们要求X在E中不是自由变量。

因为AND和OR都是满足交换律的，所以还可以使用(14.12)和(14.13)移动附加到AND或OR左操作数上的量词。例如，由(14.12)以及AND的交换律可得出如下表达式

$$\big((QX)E \text{ AND } F\big) \equiv (QX)(E \text{ AND } F)$$

这里，我们要求X在F中不是作为自由变量出现的。

✦ 示例 14.18

我们来为示例14.17中得出的重言式，也就是

$$(\forall X)p(X)\text{OR}(\exists X)\big(\text{NOT } p(X)\big)$$

变形，使得量词都在表达式之外。首先，需要为两个量词之一所使用的变量重命名。根据法则(14.6)，可以用$(\exists Y)\text{NOT } p(Y)$替代$(\exists X)\text{NOT } p(X)$，得出重言式

$$(\forall X)p(X)\texttt{OR}(\exists Y)\big(\texttt{NOT}\ p(Y)\big) \tag{14.14}$$

现在可以利用(14.13)的变形，也就是利用量词移到OR运算左操作数上的那个重言式，将∀移到OR之外，得到的表达式就是

$$(\forall X)\big(p(X)\texttt{OR}(\exists Y)\big(\texttt{NOT}\ p(Y)\big)\big) \tag{14.15}$$

表达式(14.15)与(14.14)只是形式不同，含义却没什么不同。(14.15)表述的是，对所有的X的值，以下两条中至少有一条成立：

(1) $p(X)$ 为真；

(2) 存在某个值Y，使 $p(Y)$ 为假。

要知道(14.15)为何是重言式，可以考虑对应X的某个值v。如果所考虑的解释使 $p(v)$ 为真，那么有 $p(X)\texttt{OR}(\exists Y)\big(\texttt{NOT}\ p(X)\big)$ 为真。如果 $p(v)$ 为假，那么在这一解释中，(2)肯定成立。特别要说的是，当 $Y=v$ 时，$\texttt{NOT}\ p(X)$ 为真，因此 $(\exists Y)\big(\texttt{NOT}\ p(Y)\big)$ 为真。

最后，可以应用(14.13)，将∃Y 移到OR运算之外，得到的表达式就是

$$(\forall X)(\exists Y)(p(X)\texttt{OR NOT}\ p(Y))$$

该表达式也一定是重言式。粗略地讲，它所表述的是，对每个X的值，存在某个Y的值，使得 $p(X)\texttt{OR NOT}\ p(Y)$ 为真。要知道原因，设v是X可能的取值之一。如果 $p(v)$ 在给定解释I之下为真，那么显然有如下表达式为真，不管Y是什么。

$$p(X)\texttt{OR NOT}\ p(Y)$$

如果 $p(v)$ 在解释I中为假，就可以为Y选择v，这样 $(\exists Y)\big(p(X)\texttt{OR NOT}\ p(Y)\big)$ 就为真。

14.7.6　前束式

法则(14.9)、(14.10)、(14.12)和(14.13)带来的结果是，给定任一涉及量词与逻辑运算符AND、OR和NOT的表达式，可以为其找到量词全部在外部（在表达式树的顶部）的等价表达式。也就是说，可以找到形如

$$(Q_1X_1)(Q_2X_2)\cdots(Q_kX_k)E \tag{14.16}$$

的等价表达式，其中Q_1、…、Q_k各自代表量词∀或∃中的某一个，而且子表达式E是无量词的。如此则称表达式(14.16)是*前束式*（prenex form）的。

通过以下两步，就可以把表达式变形为前束式表达式。

(1) 修正表达式。也就是说，利用法则(14.6)，使各个量词引用不同的变量，出现在一个量词中的变量既不会出现在另一个量词中，也不会作为表达式的自由变量出现。

(2) 然后，根据法则(14.9)和(14.10)把各量词移过NOT，根据法则(14.12)移过AND，并根据(14.13)移过OR。

前束式程序

原则上讲，只要重命名所有局部变量，使它们全都具有不同的变量名，然后将它们的声明移到主程序中，我们也可以把C语言程序表示为前束式程序。不过一般不会这么做，而是会选择在局部声明变量，比方说，这样一来就不必担心为在10个不同函数中用作循环指标的变量i使用各种不同的变量名了。对逻辑表达式来说，通常都有理由将表达式表示为前束式表达式，虽然这一问题超出了本书的范围。

✦ 示例 14.19

示例14.17和示例14.18都是这一过程的例子。从给出表达式 $(\forall X)p(X)\texttt{OR NOT}\big((\forall X)\ p(X)\big)$ 的示例14.17开始，通过把第二个 $\forall$ 移过NOT，就得到示例14.18中一开始的表达式

$$(\forall X)p(X)\texttt{OR}(\exists X)\big(\texttt{NOT}\ p(X)\big)$$

然后我们为用到的第二个X重命名，这是一开始就可以完成而且应该完成的。通过把两个量词移过OR运算符，就得到前束式表达式 $(\forall X)(\exists Y)\big(p(X)\texttt{OR NOT}\ p(Y)\big)$ 。

请注意，涉及AND、OR和NOT之外的逻辑运算符的表达式也可以变形为前束式表达式。正如我们在第12章中了解到的，每种逻辑运算符都可以用AND、OR和NOT表示出来。例如，$E\to F$ 就可以替换为 $\texttt{NOT}\ E\ \texttt{OR}\ F$ 。如果把各逻辑运算符都用AND、OR和NOT表示出来，就能够应用刚刚概述过的变形方式找到等价的前束式表达式。

14.7.7 量词的重新排列

最后要介绍的重言式指出了，将全称量词应用于两个变量，排列这两个量词的次序是没关系的。同样，也可以用任一次序排列两个存在量词。严格地讲就是，以下两个表达式是重言式。

$$(\forall X)(\forall Y)E\equiv(\forall Y)(\forall X)E \tag{14.17}$$

$$(\exists X)(\exists Y)E\equiv(\exists Y)(\exists X)E \tag{14.18}$$

请注意，根据(14.17)，我们能够把任一 $\forall$ 串 $(\forall X_1)(\forall X_2)(\cdots)(\forall X_k)$ 排列成选定的任意次序。事实上，(14.17)就是 $\forall$ 的交换律。同样的结论对法则(14.18)（即 $\exists$ 的交换律）也是成立的。

14.7.8 习题

(1) 将以下表达式变形为修正表达式，也就是说，任两个量词不会共用同一变量的表达式。

(a) $(\exists X)\Big(\big(\texttt{NOT}\ p(X)\big)\texttt{AND}\Big((\exists Y)\big(p(Y)\big)\texttt{OR}(\exists X)\big(q(X,Z)\big)\Big)\Big)$

(b) $(\exists X)\big((\exists X)p(X)\texttt{OR}(\exists X)q(X)\texttt{OR}\ r(X)\big)$

(2) 通过把各自由变量全称量词化，将以下表达式转换为闭表达式。如果需要的话，可以为变量重命名，使两个量词不会使用相同变量。

(a) $p(X,Y)\texttt{AND}(\exists Y)q(Y)$

(b) $(\forall X)\big(p(X,Y)\texttt{OR}(\exists X)p(Y,X)\big)$

(3) * 法则(14.12)是否暗指 $p(X,Y)\texttt{NOT}(\forall X)q(X)$ 与 $(\forall X)\big(p(X,Y)\texttt{AND}q(X)\big)$ 是等价的？对自己的回答作出解释。

(4) 把习题(1)中的表达式变形为前束式表达式。

(5) * 说明如何把量词移过→运算符。也就是说，如何把表达式 $\big((Q_1X)E\big)\to\big((Q_2Y)F\big)$ 变形为前束式表达式。大家需要对E和F中的自由变量进行什么约束？

(6) 我们可以利用重言式(14.9)和(14.10)把NOT移进量词和移出量词。利用这些法则以及德摩根律，可以移动所有的NOT，使它们直接应用到原子公式上。为下列表达式应用这种变形

(a) $\texttt{NOT}\big((\forall X)(\exists Y)p(X,Y)\big)$

(b) $\texttt{NOT}\Big((\forall X)\big(p(X)\texttt{OR}(\exists Y)q(X,Y)\big)\Big)$

(7) * 只要 $(\exists X)E$ 是重言式，E就是重言式，这样的说法是否正确？

14.8 谓词逻辑中的证明

在14.8和14.9这两节中将讨论谓词逻辑中的证明。不过，我们不会把12.11节中的分解法扩展到谓词逻辑中，虽然这样也是可行的。事实上，分解对很多使用谓词逻辑的系统来说也是极为重要的。这种证明机制将在12.10节中介绍。回顾一下，在命题逻辑的证明中，给定某些表达式 E_1、E_2、…、E_k 作为前提或者说“公理”，并构造一系列表达式（行），使各表达式符合下列条件之一。

(1) 为 E_i 之一；

(2) 根据推理规则，是用之前的0个或更多表达式得到的。

推理规则必须具有以下属性，只要我们因为 F_1、F_2、…、F_n 出现在表达式列中，而可以向表达式列中添加F，就有

$$(F_1 \text{AND } F_2 \text{AND} \cdots \text{AND} F_n) \to F$$

是重言式。

谓词逻辑中的证明基本是相同的。当然，作为前提和证明中各行的表达式都是谓词逻辑表达式，而非命题逻辑表达式。此外，一个表达式的自由变量与另一个表达式中同名的自由变量是不能存在关系的，所以会要求前提和证明中各行都是闭公式。

14.8.1 隐式全称量词

然而，一般约定在书写证明中的表达式时，不会显式给出最外层的全称量词。例如，考虑示例14.3中的表达式

$$\big(csg(\text{“CS101”}, S, G) \text{AND } snap(S, \text{“C.Brown”}, A, P)\big) \to answer(G) \tag{14.19}$$

表达式(14.19)可能是证明的某一前提。在示例14.3中，我们凭直觉将其看作谓词*answer*的定义。比方说，我们在*answer* ("A")，也就是C.Brown的CS101课程得了A的证明中可能用到(14.19)。

在示例14.3中，对(14.19)含义的解释是，对所有的S、G、A和P的值，如果说学号为S的学生CS101课程得到了成绩G，也就是说，如果 $csg(\text{"CS101"}, S, G)$ 为真，而且学号为S的学生名叫C.Brown，地址为A，电话号码为P，也就是说，如果 $snap(S, \text{"C.Brown"}, \text{A}, \text{P})$ 为真，那么G就是一种答案，即*answer* (G)为真。在示例14.3的那个例子中，我们还没有正式的量词概念。不过，现在看到，真正想要断言的是

$$(\forall S)(\forall G)(\forall A)(\forall P)\Big(\big((csg(\text{“CS101”}, S, G) \text{AND } snap(S, \text{“C.Brown”}, A, p)\big) \to answer(G)\Big)$$

因为经常需要在表达式前引入一串全称量词，所以我们会用简化符号 $(\forall *)E$ 表示一串全称量词 $(\forall X_1)(\forall X_2)\cdots(\forall X_k)E$，其中 X_1、X_2、…、X_k 是表达式E的自由变量。例如，(14.19)就可以写成

$$(\forall *)\Big(\big(csg(\text{"CS101"}, S, G) \text{AND } snap(S, \text{"C.Brown"}, A, P)\big) \to answer(G)\Big)$$

不过，我们将继续把E中的自由变量称为 $(\forall *)E$ 中的“自由变量”。这样使用术语“自由”严格来讲是不正确的，但是非常实用。

14.8.2 作为推理规则的变量替换

除了第12章中讨论过的对应命题逻辑的推理规则外，比如肯定前件式假言推理，以及在证明的前一行中进行以相等换相等的替换，对谓词逻辑中的证明来说还有一种特别实用的涉及变

量替换的推理规则。如果已经断言作为前提或证明中某一行的表达式E，而且E'是通过用变量或常量替换E中某个自由变量形成的表达式，那么$E \to E'$是重言式，而且我们可以向证明中添加E'这样一行。务必记住，我们不能替换E的约束变量，只能替换E的自由变量。

严格地讲，可以用函数sub作为表达式变量的替换。对E中各自由变量X而言，可以定义$sub(X)$为某个变量或某个常量。如果没有为$sub(X)$指定值，就要假设想要的是$sub(X) = X$。如果E是任一谓词逻辑表达式，表达式$sub(E)$就是将E中所有作为自由变量出现的X用$sub(X)$替代后得到的。

证明中出现的表达式

请记住，如果在证明中看到表达式E，它其实是$(\forall *)E$的简略形式。要注意到$E \equiv (\forall *)E$一般不是重言式，因此我们显然是在用一个表达式代表一个与之不同的表达式。

还要记住，当证明中出现E时，并不是在断言$(\forall *)E$是重言式，而是在断言$(\forall *)E$是根据前提得出的。也就是说，如果E_1、E_2、…、E_n是前提，而且正确书写了证明中的行E，则可知

$$\big((\forall *)E_1 \text{AND} (\forall *)E_2 \text{AND} \cdots \text{AND} (\forall *)E_n\big) \to (\forall *)E$$

是重言式。

变量替换法则说明$E \to sub(E)$是重言式。因此，如果E是证明中的行，就可以在同一证明中加入$sub(E)$这一行。

✦ 示例 14.20

考虑以表达式(14.19)

$$\big(csg(\text{“CS101”}, S, G) \text{AND } snap(S, \text{“C.Brown”}, A, P)\big) \to answer(G)$$

作为E。可以将一种可能的替换sub定义为

$$sub(G) = \text{"}B\text{"}$$
$$sub(P) = S$$

也就是说，这里要用常量B替换变量G，并用变量S替换变量P。而变量S和A保持不变。表达式$sub(E)$就是

$$\big(csg(\text{“CS101”}, S \text{“}B\text{”}) \text{AND } snap(S, \text{“C.Brown”}, A, S)\big) \to answer(\text{“}B\text{”}) \qquad (14.20)$$

通俗地说，表达式(14.20)的意思是，如果学生S的CS101课程得了B，该学生的姓名是C. Brown，而且该学生的电话号码和学号是唯一的，那么“B”就是答案。

请注意，(14.19)表达了更具一般性的规则，而(14.20)只是它的一个特例。也就是说，只有在成绩为B，而且C. Brown的学号巧合到与他的电话号码相同时，(14.20)才能推理出正确答案，否则(14.20)不说明任何问题。

✦ 示例 14.21

表达式

$$p(X,Y) \text{OR} (\exists Y) q(X,Z) \qquad (14.21)$$

中含有自由变量X和Y，以及约束变量Z。回想一下，从技术上讲，(14.21)其实代表闭表达式$(\forall *)\big(p(X,Y)\big) \text{OR} (\exists Z) q(X,Z)$，而这里的$(\forall *)$代表对自由变量$X$和$Y$的量化，也就是有

$$(\forall X)(\forall Y)\big(p(X,Y)\big)\text{OR}(\exists Z)q(X,Z)$$

在(14.21)中，可以替换 $sub(X)$ ＝a和 $sub(Y)$ ＝b，得到表达式 $p(a,b)\text{OR}(\exists Z)q(a,Z)$ 。不难看出，这一不含自由变量（因为我们用常量替换了其中各自由变量）的表达式是(14.21)的一个特例，它陈述了要么 $p(a,b)$ 为真，要么存在某个Z的值，使得 $p(a,Z)$ 为真。正式地讲，

$$\Big((\forall X)(\forall Y)\big(p(X,Y)\text{OR}(\exists Z)q(X,Z)\big)\Big)\rightarrow\big(p(a,b)\text{OR}(\exists Z)q(a,Z)\big)$$

是重言式。

有人可能想知道在用a和b替换X和Y时(14.21)中隐含的量词会发生什么。答案是，得到的表达式 $p(a,b)\text{OR}(\exists Z)q(a,z)$ 中不存在自由变量，隐含的表达式 $(\forall^*)\big(p(a,b)\text{OR}(\exists Z)q(a,z)\big)$ 没有全称量词前缀，也就是说

$$p(a,b)\text{OR}(\exists Z)q(a,Z)$$

在这种情况中只代表自身。我们不会把 $(\forall^*)$ 替换为 $(\forall a)(\forall b)$ ，因为常量不可能被量词化，这样做是行不通的。

替换的特例

示例14.20是一般情况，其中只要我们对表达式E应用替换sub，得到的就是E的特例。如果sub用常量c替换变量X，那么表达式 $sub(E)$ 就只适用于 $X=c$ 的情况，而不适用于其他情况。如果sub让两个变量变得相同，那么 $sub(E)$ 就只适用于两个变量具有相同值的特例。然而，变量的替换往往正是我们进行证明时所需的，因为它们让我们可以在特例中应用一般规则，而且让我们可以将规则组合起来构成其他规则。我们将在14.9节中研究这种形式的证明。

14.8.3 习题

根据前提，利用12.10节中讨论过的推理规则以及刚刚讨论过的变量替换规则，证明以下结论。请注意，大家可以把任何命题或谓词演算的重言式用作证明的某一行。不过，请尽量只使用12.8节、12.9节和14.7节中介绍的重言式。

(a) 根据前提 $(\forall X)p(X)$ ，证明结论 $(\forall X)p(X)\text{OR } q(Y)$ 。

(b) 根据前提 $(\exists X)p(X,Y)$ ，证明结论 $\text{NOT}\Big((\forall X)\big(\text{NOT } p(X,a)\big)\Big)$ 。

(c) 根据前提 $p(X)$ 和 $p(X)\rightarrow q(X)$ ，证明结论 $q(X)$ 。

14.9 根据规则和事实的证明

谓词逻辑中形式最简单的证明可能涉及以下两类前提。

(1) 事实（fact），它们是基本原子公式。

(2) 规则（rule），它们是“如果–那么”形式的表达式。我们在示例14.20中讨论过的有关C.Brown在课程CS101所取得成绩的查询(14.19)就是一个例子

$$\big(csg(\text{“CS101”},S,G)\text{AND } snap(S,\text{“C.Brown”},A,P)\big)\rightarrow answer(G)$$

规则由蕴涵符号左侧的一个或更多原子公式的AND以及蕴涵符号右侧的一个原子公式组成。假设出现在右部的任何变量也会出现在左部中的某处。

规则的左边（前提）叫作*左部*（body），而右边叫作*右部*（head）。左部中的任一原子公式

叫作子目标（subgoal）。例如，在上面再次表示出的规则(14.19)中，子目标是 $csg("CS101", S, G)$ 和 $snap(S, "C.Brown", A, P)$ 。右部是 $answer(G)$ 。

规则的一般使用思路是，规则是可以应用于事实的一般原则。我们会试着通过替换规则中的变量，将规则左部的子目标与给定或已经证明的事实匹配起来。如果能这样做，这种替换会使右部成为基本原子公式，因为已经假设右部的各变量都会出现在左部中。我们可以把这一新的基本原子公式添加到自己可支配的事实集中，以作进一步证明之用。

✦ 示例 14.22

根据规则和事实的证明有一种简单的应用，就是用于回应第8章讨论的关系模型中的查询。关系对应的是谓词符号，而关系中的元组则对应着基本原子公式，这些基本原子公式具有谓词符号以及依次等同于元组组分的参数。例如，由图8-1中的“课程–学号–成绩”关系，可以得到如下事实

$$csg(\text{“CS101”}, 12345, \text{“A”}) \qquad csg(\text{“CS101”}, 67890, \text{“B”})$$
$$csg(\text{“EE200”}, 12345, \text{“C”}) \qquad csg(\text{“EE200”}, 22222, \text{“B+”})$$
$$csg(\text{“CS101”}, 33333, \text{“A–”}) \qquad csg(\text{“PH100”}, 67890, \text{“C+”})$$

同样，由图8-2a中的“学号–姓名–地址–电话”关系，可以得到以下事实

$$snap(12345, \text{“C.Brown”}, \text{“12Apple St.”}, 555-1234)$$
$$snap(67890, \text{“L.Van Pelt”}, \text{“34Pear Ave.”}, 555-5678)$$
$$snap(22222, \text{“P.Patty”}, \text{“56Grape Blvd.”}, 555-9999)$$

对这些事实，可以添加规则(14.19)

$$\left(csg\left(\text{“CS101”}, S, G\right) \text{AND} snap\left(S, \text{“C.Brown”}, A, P\right)\right) \to answer\left(G\right)$$

从而完成前提列表。

假设想要证明 $answer("A")$ 为真，也就是说，C.Brown的CS101课程得了A。在证明开头部分要列出所有的事实和规则，虽然在这里我们只需要规则、第一条 *csg* 事实和第一条 *snap* 事实即可。证明的前3行就是

(1) $\left(csg\left(\text{“CS101”}, S, G\right) \text{AND} snap\left(S, \text{“C.Brown”}, A, P\right)\right) \to answer\left(G\right)$

(2) $csg(\text{“CS101”}, 12345, \text{“A”})$

(3) $snap(12345, \text{“C.Brown”}, \text{“12Apple St.”}, 555-1234)$

下一步是利用推理规则（如果 E_1 和 E_2 是证明中某两行，则可以把 E_1 AND E_2 写为证明中的一行）把第二行和第三行结合起来，因此就有了这样一行

(4) $csg\left(\text{“CS101”}, 12345, \text{“A”}\right) \text{AND}$

$snap(12345, \text{“C.Brown”}, \text{“12Apple St.”}, 555-1234)$

然后，利用自由变量的替换法则特化我们的规则——第(1)行，使它适用于第(4)行中的常量。也就是说，要在第(1)行中进行以下替换

$$sub\left(S\right) = \text{“CS101”}$$
$$sub\left(G\right) = \text{“A”}$$
$$sub\left(A\right) = \text{“12Apple St.”}$$
$$sub\left(P\right) = 555-1234$$

得到如下一行

(5) $csg(\text{“CS101”}, 12345, \text{“A”})$ AND $snap(12345, \text{“C.Brown”}, \text{“12Apple St.”}, 555-1234)$
$\rightarrow answer(\text{“A”})$

最后，对(4)和(5)应用肯定前件式假言推理就得到该证明的第六行也是最后一行

(6) $answer(\text{“A”})$ 。

14.9.1 简化的推理规则

如果看过示例14.22中的证明，就可以得出以下根据基本原子公式和逻辑规则构建证明的策略。

(1) 选择一条要应用的规则，并选择一种替换，将各个子目标转换成基本原子公式，这些基本原子公式要么是给定的事实，要么是已经证明的内容。在示例14.22中，我们用12345替换了S，等等。得到的结果就如示例14.22的第(4)行所示。

(2) 为替换过的各个子目标创建证明中的行，要么因为这些子目标是事实，要么用某种方式推断出它们。这一步是示例14.22中的第(2)行和第(3)行。

(3) 创建一行，表示与替换过的各子目标对应的那几行的AND。这一行是替换过规则的左部。在示例14.22中，这一步出现在第(5)行。

(4) 利用肯定前件式假言推理、第(3)步替换过的左部，以及第(1)步替换过的规则，推论出替换过的右部。这一步就是示例14.22的第(6)行。

可以按照如下方式把这些步骤结合成一条推理规则。如果在前提中存在规则R，而且存在替换sub满足在替换过的实例$sub(R)$中，各子目标都是证明中的行，就可以把$sub(R)$的右部添加为证明中的一行。

规则的诠释

和所有出现在证明中的表达式一样，规则也是因式全称量词化的。因此可以把(14.19)说成是“对所有的S、G、A和P，如果$csg(\text{"CS101"}, S, G)$为真，而且$snap(S, \text{"C.Brown"}, A, P)$为真，那么$answer(G)$为真”。不过，可以将出现在左部中但未出现在右部中的变量，比如S、A和P，当作左部范围内的存在量词。正式地讲就是(14.19)等价于

$$(\forall G)\Big(\big((\exists S)(\exists A)(\exists P)\big(csg(\text{"CS101"}, S, G) \text{ AND } snap(S, \text{"C.Brown"}, A, P)\big) \rightarrow answer(G)\Big)$$

也就是说，对所有的G，如果存在S、A和P满足$csg(\text{"CS101"}, S, G)$和$snap(S, \text{"C.Brown"}, A, P)$都为真，$answer(G)$就为真。

这种说法更接近我们应用规则的方式。它表示，对右部中出现的变量的各个值，我们应该试着找出只出现在左部中的变量使得左部为真时的值。如果找到这样的值，则右部对所选的右部变量的值而言为真。

要知道为什么可以把只出现在左部中的变量当作存在量词化的变量，首先从形如$B \rightarrow H$这样的规则开始，其中B是左部，H是右部。设X是只出现在B中的变量。这一规则隐含着如下形式

$$(\forall *)(B \rightarrow H)$$

而且根据法则(14.17)，可以让对应X的量词位于最内侧，将表达式写为$(\forall *)(\forall X)(B \rightarrow H)$。在此，$(\forall *)$包含了除$X$之外的所有变量。现在可以利用只使用NOT和OR的等价表达式，即

$(\forall *)(\forall X)\big((\text{NOT } B)\text{OR } H\big)$ 来代替蕴涵式了。因为X没有出现在H中，所以可以反向应用法则(14.13)，从而让$(\forall X)$只应用于NOT B，得到$(\forall *)\big(((\forall X)\text{NOT } B)\text{OR } H\big)$。接下来，利用法则(14.10)把$(\forall X)$移动到否定内，就得出

$$(\forall *)\big((\text{NOT}(\exists X)(\text{NOT NOT } B))\text{OR } H\big)$$

消除双重否定之后就是$(\forall *)\big((\text{NOT}(\exists X)B)\text{OR } H\big)$。最后，还原蕴涵就得到$(\forall *)\big(((\exists X)B)\rightarrow H\big)$。

✦ 示例 14.23

在示例14.22中，规则R是(14.19)，或者说

$$\big(csg(\text{"CS101"},S,G)\text{ AND } snap(S,\text{"C.Brown"},A,P)\big)\rightarrow answer(G)$$

替换sub就如示例14.22中给出的那样，而且$sub(R)$的子目标是示例14.22中的第(2)和第(3)行。根据新的推理规则，可以立即写出示例14.22的第(6)行，而不需要第(4)行和第(5)行。其实，只要第(1)行（规则R本身）是给定的前提，就可以从证明中省略掉它。

✦ 示例 14.24

再举个规则在证明中如何应用的例子。考虑一下图8-2b中的“课程–前提”关系，其中的8个事实可以用如下8个具有谓词cp的基本原子公式表示。

$$cp(\text{“CS101”},\text{“CS100”}) \quad cp(\text{“EE200”},\text{“EE005”})$$
$$cp(\text{“EE200”},\text{“CS100”}) \quad cp(\text{“CS120”},\text{“CS101”})$$
$$cp(\text{“CS121”},\text{“CS120”}) \quad cp(\text{“CS205”},\text{“CS101”})$$
$$cp(\text{“CS206”},\text{“CS121”}) \quad cp(\text{“CS206”},\text{“CS205”})$$

我们可能希望另一个谓词$before(X,Y)$表示在选修课程X之前必须修完课程Y。课程Y可能是X的前提，或是X前提的前提，诸如此类。可以通过以下描述递归地定义“之前”的概念。

(1) 如果Y是X的前提，那么Y在X之前。

(2) 如果Z是X的前提，而且Y在Z之前，那么Y在X之前。

规则(1)和(2)可以表示为如下的谓词逻辑规则。

$$cp(X,Y)\rightarrow before(X,Y) \tag{14.22}$$

$$\big(cp(X,Z)\text{ AND } before(Z,Y)\big)\rightarrow before(X,Y) \tag{14.23}$$

现在要来探索一些根据本示例开头部分给出的8条“课程–前提”事实以及规则(14.22)和(14.23)可以证明的$before$事实。首先，可以应用规则(14.22)把各条cp事实转换成相应的$before$事实，得到：

$$before(\text{“CS101”},\text{“CS100”}) \quad before(\text{“EE200”},\text{“EE005”})$$
$$before(\text{“EE200”},\text{“CS100”}) \quad before(\text{“CS120”},\text{“CS101”})$$
$$before(\text{“CS121”},\text{“CS120”}) \quad before(\text{“CS205”},\text{“CS101”})$$
$$before(\text{“CS206”},\text{“CS121”}) \quad before(\text{“CS206”},\text{“CS205”})$$

例如，可以对(14.22)使用替换

$$sub_1(X) = \text{“CS101”}$$
$$sub_1(Y) = \text{“CS100”}$$

得到替换过的规则实例

$$cp(\text{“CS101”}, \text{“CS100”}) \rightarrow before(\text{“CS101”}, \text{“CS100”})$$

这条规则加上前提 $cp(\text{“CS101”}, \text{“CS100”})$，就给出了

$$before(\text{“CS101”}, \text{“CS100”})$$

现在可利用规则(14.23)、前提 $cp(\text{“CS101”}, \text{“CS100”})$ 以及刚证明的事实 $before(\text{“CS101”}, \text{“CS100”})$，证明

$$before(\text{“CS120”}, \text{“CS100”})$$

也就是，对(14.23)应用替换

$$sub_2(X) = \text{“CS120”}$$
$$sub_2(Y) = \text{“CS100”}$$
$$sub_2(Z) = \text{“CS101”}$$

得到规则

$$\left(cp(\text{“CS120”}, \text{“CS101”}) \text{ AND } before(\text{“CS101”}, \text{“CS100”})\right) \rightarrow before(\text{“CS120”}, \text{“CS100”})$$

然后可以推断出这一替换过的规则的右部，从而证明

$$before(\text{“CS120”}, \text{“CS100”})$$

同样，可以对基本原子公式

$$cp(\text{“CS121”}, \text{“CS120”})$$

和 $before(\text{“CS120”}, \text{“CS100”})$ 应用规则(14.23)，证明 $before(\text{“CS121”}, \text{“CS100”})$。然后对基本原子公式 $cp(\text{“CS206”}, \text{“CS121”})$ 和 $before(\text{“CS121”}, \text{“CS100”})$ 应用规则(14.23)，证明

$$before(\text{“CS206”}, \text{“CS100”})$$

还有若干条其他的*before*事实也可以通过同样的方式来证明。

图中的路径

示例14.24所处理的，是规则的一种常见形式，它在给定有向图中弧的情况下，定义了图中的路径。把课程当作节点，如果课程*b*是课程*a*的前提，就存在弧 $a \rightarrow b$。那么*before*(*a*,*b*)就对应着从*a*到*b*的某一条长度为1或更长的路径。图14-9展示了基于图8-2b中“课程–前提”信息绘制的有向图。

在图表达是前提时，我们希望它是无环的，因为选修某门课程的前提是修过这门课本身。不过，即便图中含有环路，同类的逻辑规则也仍然用弧定义了路径。可以把这些规则写成

$$arc(X,Y) \rightarrow path(X,Y)$$

也就是说，如果存在从节点*X*到节点*Y*的弧，就存在从*X*到*Y*的路径，而且

$$\left(arc(X,Z) \text{ AND } path(Z,Y)\right) \rightarrow path(X,Y)$$

也就是说，如果存在从*X*到某节点*Z*的弧，而且存在从*Z*到*Y*的路径，那么存在从*X*到*Y*的路

径。请注意，这些内容与(14.22)和(14.23)表示的是同样的规则，其中用谓词*arc*代替了*cp*，用*path*代替了*before*。

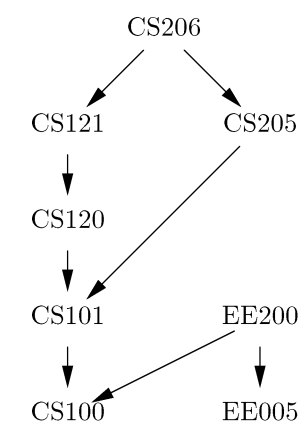


图14-9　表示成有向图的前提信息

14.9.2　习题

(1) * 我们可以按照以下方式证明示例14.24中的谓词*before*是谓词*cp*的传递闭包。假设存在一系列的课程c_1、c_2、…、c_n，其中$n \geqslant 2$，而且c_1是c_2的前提，c_2是c_3的前提，以此类推，对$i=1$、2、…、$n-1$而言，$cp(c_i, c_{i+1})$是给定的事实。通过对*i*的归纳证明对$i=2$、3、…、*n*，有$before(c_1, c_i)$，从而证明c_1在c_n之前。

(2) 利用示例14.24中的规则和事实，证明以下事实。

(a) *before*(“CS120”,“CS100”)

(b) *before*(“CS206”,“CS100”)

(3) 通过向示例14.24添加规则

$$\big(before(X,Z)\ \text{AND}\ before(Z,Y)\big) \rightarrow before(X,Y),$$

可以提高处理前提链的速度。也就是说，第一个子目标可以是任一*before*事实，而不止是前提事实。利用该规则，为习题(2)的(b)小题找出更简短的证明。

(4) * 示例14.24中有多少*before*事实是可以证明的？

(5) 设*csg*是代表图8-1中“课程–学号–成绩”关系的谓词，*cdh*是代表图8-2c中课程–日子–时刻关系的谓词，*cr*是代表图8-2d中“课程–教室”关系的谓词。并设*where*(*S*,*D*,*H*,*R*)是表示学号为*S*的学生*D*那天*H*时在教室*R*上课。更精确地讲，学号为*S*的学生选修的课程是那个时间在那个教室上课。

(a) 写出用*csg*、*cdh*和*cr*定义*where*的规则。

(b) 如果对应谓词*csg*、*cdh*和*cr*的事实是图8-1和图8-2中给出的，那么可以推导出哪些*where*事实？证明两条这样的事实。

14.10　真理和可证性

在为谓词逻辑的讨论收尾时，我们要介绍一个更为微妙的逻辑问题：可证明内容与真实内容之间的区别。前面已经看到过这样一些推理规则，它们允许我们证明命题逻辑或谓词逻辑中的内容，但我们不确定给定的规则集合是否完备，是否允许我们证明每一个真命题。例如，我

们断言12.11节中所表示的分解对命题逻辑而言是完备的。分解的一种一般化形式（这里没有涵盖）对谓词逻辑而言也是完备的。

14.10.1　模型

不过，要理解证明策略的完备性，就需要掌握“真理”的概念。要发现“真理”，需要理解模型（model）的概念。每种逻辑对表达式集而言都有模型的概念，这些模型是令表达式为真的解释。

✦　示例 14.25

在命题逻辑中，解释是真值赋值。考虑表达式 $E_1 = p \text{ AND } q$ 和 $E_2 = \overline{p} \text{ OR } r$。涉及变量$p$、$q$和$r$的表达式有8种真值赋值，我们可以用依次对应各变量真值的3位构成的位串来表示这些真值赋值。

只有真值赋值令p和q都为真（即110和111这两种真值赋值）时，它才能令表达式 E_1 为真。而000、001、010、011、101和111这6种真值赋值可以让表达式 E_2 为真。因此只有一种对应表达式集 $\{E_1, E_2\}$ 的模型，即111，因为只有该模型出现在两个列表中。

✦　示例 14.26

在谓词逻辑中，解释是14.5节中定义过的结构。我们来考虑表达式E

$$(\forall X)(\exists Y)p(X,Y)$$

也就是说，对每个X的值来说，至少存在一个Y值，使得 $p(X,Y)$ 为真。

如果对定义域D中每个元素a来说，在D中存在某个元素b（对各个a不一定有相同的b），使得“谓词p的解释”这个关系中具有成员“有序对(a,b)”，就说解释使得E为真。这些解释就是E的模型，其他的解释则不是。例如，如果定义域D是整数，而且当且仅当 $X < Y$ 时，谓词p的解释使得 $p(X,Y)$ 为真，我们就有了对应表达式E的模型。不过，定义域D也是整数，而且p的解释是当且仅当 $X = Y^2$ 时 $p(X,Y)$ 为真，这一解释就不是表达式E的模型。

14.10.2　蕴涵

在给定表达式集 $\{E_1, E_2, \cdots, E_n\}$ 的情况下，就可以陈述表达式E为真的含义了。如果每个对应 $\{E_1, E_2, \cdots, E_n\}$ 的模型M也是对应E的模型，就说 $\{E_1, E_2, \cdots, E_n\}$ 蕴涵（entail）表达式E。双十字转门（double turnstile）运算符⊨就表示蕴涵，如

$$E_1,\ E_2,\cdots,\ E_n \models E$$

我们需要凭直觉意识到每种解释都是一个可能存在的世界。当说 $E_1,\ E_2,\cdots,\ E_n \models E$ 时，也就是在说，在使表达式 $E_1,\ E_2,\cdots,\ E_n$ 为真的每个可能存在的世界中，E都为真。

蕴涵的概念应该是与证明的概念形成对照的。如果我们有某个特定的证明系统，比如说分解证明，那么可以使用单十字转门（single turnstile）运算符⊢用同样的方式表示证明。也就是说，

$$E_1,\ E_2,\cdots,\ E_n \vdash E$$

意味着，对当前所拥有的推理规则集而言，存在从前提 $E_1,\ E_2,\cdots,\ E_n$ 到E的证明。请注意，⊢运算符对不同的证明系统可能存在不同的含义。还要记住，虽然我们一般乐于使用当且仅当另一

者为真时一者为真的证明系统，但是$\vDash$和$\vdash$不一定是相同的关系。

重言式与蕴涵之间有着密切联系。特别要指出的是，假设$E_1, E_2, \cdots, E_n \vDash E$。然后可以声明

$$(E_1 \text{ AND } E_2 \text{ AND} \cdots \text{AND } E_n) \rightarrow E \tag{14.24}$$

是重言式。若某解释I使(14.24)左边为真，则I是$\{E_1, E_2, \cdots, E_n\}$的模型。因为$E_1, E_2, \cdots, E_n \vDash E$，所以$I$一定也能使$E$为真。因此$I$使(14.24)为真。

其他的可能就只有I使(14.24)的左边为假。这样一来，因为当蕴涵的左边为假时它恒为真，可知(14.24)还是为真。因此(14.24)是重言式。

反过来，如果(14.24)是重言式，则可以证明$E_1, E_2, \cdots, E_n \vDash E$。这里把这一证明留作本节习题。

请注意，我们的论证并不取决于涉及的表达式是命题逻辑表达式还是谓词逻辑表达式，或者是我们没有了解的某种其他逻辑的表达式。只需要知道，重言式是指那些所有“解释”都使之为真的表达式，而表达式或表达式集的模型是指令这一或这些表达式为真的某种解释。

14.10.3　可证性与蕴涵的比较

我们想要给定的证明系统允许我们证明所有为真的事物，而且不会证明为假的事物。也就是说，我们希望单十字转门和双十字转门符号表示相同的含义。如果只要某事物可证，它也就被蕴涵，那么该证明系统是相容的（consistent）。也就是说，$E_1, E_2, \cdots, E_n \vDash E$蕴涵$E_1, E_2, \cdots, E_n \vdash E$。例如，我们在12.10节中讨论过为什么命题逻辑的推理规则是相容的。准确地讲，我们证明了，只要从前提E_1、E_2、$\cdots$、E_n开始，并在证明中写出一行E，就有$(E_1 \text{ AND } E_2 \text{ AND} \cdots \text{AND } E_n) \rightarrow E$是重言式。根据上面论证过的，这就等同于表述$E_1, E_2, \cdots, E_n \vDash E$。

我们还希望证明系统是完备的。这样就可以证明所有由前提蕴涵的事物，即便要找到该证明是很难的。事实证明，12.10节给出的推理规则，还有12.11节的分解规则都是完备证明系统。也就是说，如果$E_1, E_2, \cdots, E_n \vDash E$，则$E_1, E_2, \cdots, E_n \vdash E$也在这些证明系统中。完备的谓词逻辑证明系统是存在的，虽然我们不会介绍它们。

14.10.4　哥德尔不完备性定理

这个现代数学最引人注目的结论之一通常被看作与我们刚说过的谓词逻辑存在完备证明系统是矛盾的。事实上，这一结论并未涉及前面讨论过的谓词逻辑，而是关系到谓词逻辑的一种特殊形式，这种逻辑只谈论整数以及常见的整数运算。特别要说的是，我们必须对谓词逻辑加以修改，从而引入对应算术运算的谓词，比如

(1) $plus(X, Y, Z)$，我们希望当且仅当$X + Y = Z$时它才为真，

(2) $times(X, Y, Z)$，只有在$X \times Y = Z$时为真，以及

(3) $less(X, Y)$，只有在$X < Y$时为真。

此外，我们需要对解释中的定义域加以限制，以使这些值都是非负整数。完成这一工作有两种方式。一种是引入由我们断言为真的表达式构成的表达式集。通过恰当地选择这些表达式，任何满足表达式的解释中的定义域都必须“看似”整数，而且像*plus*或*less*这样的特殊谓词必须具有和同名运算相同的行为。

✦ 示例 14.27

我们可以断言像

$$plus(X,Y,Z) \to plus(Y,X,Z)$$

这样表示加法交换律的表达式，或表示<关系传递性的表达式

$$\big(less(X,Y)\ \text{AND}\ less(Y,Z)\big) \to less(X,Z)$$

也许理解哥德尔的定理为谓词逻辑带来的限制有种更简单的方式，就是假设这种逻辑只允许一种模型，这种模型的定义域是非负整数，而且为这些特殊的谓词给定了与它们的约定含义对应的关系。例如，我们可以令对应谓词*plus*的解释为

$$\{(a,b,c)\,|\,a+b=c\}$$

哥德尔的定理说的是，不管选择什么相容的证明系统，总是存在某个为真但不可证明的表达式E！更精确地讲，如果E_1、E_2、…、E_n是其模型相当于非负整数的这样一些表达式，那么有E_1，E_2,…，$E_n \vDash E$为真，但E_1，E_2,…，$E_n \vdash E$为假。也就是说，在我们选定的证明系统中，不可能根据$\{E_1,E_2,\cdots,E_n\}$证明E。

对选定的不同证明系统而言，不可证明的表达式E可能是不同的。事实上，所选的表达式E可被视作把该表达式本身在给定证明系统中不可证明这一事实编码为整数的一种方法。

14.10.5 计算机能完成的工作的限制

哥德尔的理论表明我们回答与数学相关的问题的能力是有限的。如果拥有像整数这样复杂的数学模型，以及我们在本书中已经见识过的，比整数还要复杂得多的数学模型，就不存在将真命题与假命题区分开的机械方式。我们能做到的最多也就是利用某种相容的证明系统以便可以寻找证明。如果找到一种，就算是非常幸运了，而且可以确定被证明的命题为真。不过，这种找寻可能一直继续下去，而永远都找不到证明，即便该命题为真。也就是说，我们定义手头的数学模型时所做的假设蕴涵了该命题。

从哲学的角度讲，这种情况说明了数学永远会是充满乐趣和挑战的。从实践的角度讲，它表明用计算机可以完成的任务是有限的。特别要指出的是，虽然可以编写程序在简单的系统（比如命题逻辑或不含任何特殊谓词或限制的谓词逻辑）中寻找证明，但没法为足够复杂的系统完成这一工作。大家应该看看下文附注栏内容“不可判定性”，这里讨论了一个与哥德尔不完备性定理有关的定理。不可判定性的理论让我们可以指出一些可证明计算机无法解决的具体问题。

不可判定性

逻辑学家阿兰·图灵（Alan Turing）在20世纪30年代就创立了正式的计算理论，这要比根据他的理论构造的电子计算机早出现很久。这一理论最重要的结果就是发现某些问题是不可判定的（undecidable），无论什么计算机都不能解答这些问题。

该理论最重要的特征是图灵机（Turing machine），一种由有限自动机和被分成无限个方格的纸带组成的抽象计算机。在一次移动中，图灵机可以读它的读写头（tape head）看到的那个方格中所含的字符，并根据这个字符以及图灵机的当前状态，用另一个字符代替该字符，改变图灵机的状态，并将读写头左移或右移一格。很明显，所有真实的计算机，以及其他那些研究计算机应该是怎样的数学模型，都刚好能计算图灵机可以计算的内容。因此可以把图灵机当作计算机的标准抽象模型。

不过，我们不必详细了解图灵机是怎样给图灵的理论增色的。只要拿它作为计算机模型，那种读字符输入并且只有两种可能的写语句`printf("yes\n")`和`printf("no\n")`的C语

言程序，就足够了。此外，在产生任一种输出后，该程序必须终止，这样它才不会随后产生矛盾的输出。要理解，这类程序在接受某些输入后可能既不能回应“yes”也不会回应“no”，而是在循环中一直循环下去。

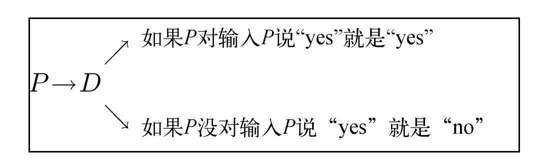
我们将要证明，不存在像图14-10a所示的“判定器”程序D这样的程序。假设D接受上述特殊类型的程序P作为输入，在给定P本身作为P的输入时，若P说“yes”，则D说“yes”。而在给定P作为P的输入时，若P说“no”或者P没法进行任何判定，则D说“no”。正如我们将要看到的，正是这一要求使得D可以算出什么时候P从不会作出使得D无法写的判定。

不过，假设这样的D存在，要编写如图14-10b所示“求补器”程序C就是个简单问题了。C是根据假设的D，通过把所有打印“no”的语句变成打印“yes”的语句，并把打印“yes”的语句变成打印“no”的语句而形成的程序。

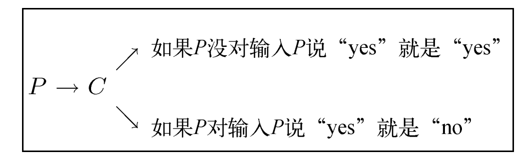
现在可以询问，如图14-10c所示，在把C本身提供给C作为输入时，会发生什么？如果C说“yes”，那么按照图14-10b表示的，C会断言“C针对输入C不会说‘yes’”。如果C说“no”，那么C会断言“C针对输入C会说‘yes’”。现在就有了类似罗素悖论的矛盾，其中C没法真实地说出“yes”和“no”。

结论就是，这样的判定器程序D其实是不存在的。也就是说，D可以解决的问题，也就是在给定其自身作为输入时类型限制为说“yes”或没法说“yes”（通过说“no”或什么都不说）的C语言程序，是不能由计算机解决的。它是个不可判定的问题。

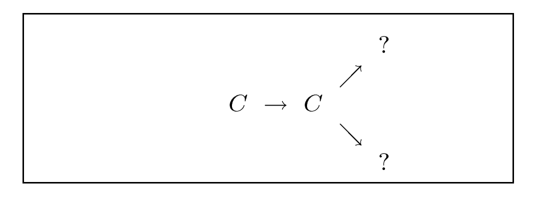
自图灵最初的结果起，现已发现种类繁多的不可判定问题。例如，某给定输入是否会让给定的C语言程序进入无限循环，或两个C语言程序在接受相同输入时是否会产生相同的输出，都是不可判定的。



(a) 假设的判定器D



(b) 求补器C



(c) 在以其自身作为输入时，C会做些什么？

图14-10　图灵构造的一部分

因此，本书并不是要以负面的声音收尾，而是要用不可判定性这一理论提醒我们，和数学一样，计算机科学也注定是挑战最优秀人才的。深入研究这一学科的学生还将了解到避免这种不可判定（而且难以驾驭）的情况所需要的科学与艺术。这些学生之后可能会加入到推进计算机发展的科学家和工程师的队伍之中。

14.10.6 习题

(1) 设 $E_1 = p$ ，$E_2 = q$ ，而且 $E_3 = qr + p\bar{r}$ 。描述使$\{E_1,E_2\}$为真的模型（命题变量p、q和r的真值赋值）。描述 E_3 的模型。$E_1, E_2 \vDash E_3$ 是否为真？为什么？

(2) ** 考虑以下谓词逻辑表达式。

a) $E_1 = (\forall X)p(X,X)$

b) $E_2 = (\forall X)(\forall Y)\big(p(X,Y) \rightarrow p(Y,X)\big)$

c) $E_3 = (\forall X)(\forall Y)(\forall Z)\Big(\big(p(X,Y)\text{ AND } p(Y,Z)\big) \rightarrow p(X,Z)\Big)$

d) $E_4 = (\forall X)(\forall Y)\big(p(X,Y)\text{ OR } p(Y,X)\big)$

e) $E_5 = (\forall X)(\exists Y)p(X,Y)$

这5个表达式中哪个表达式是由其他4个表达式蕴涵的？在各情况中，要么给出与所有可能的解释有关的论证以证明这种蕴涵，要么给出可以作为其中4个表达式的模型但不是第5个表达式的模型的某种特定解释。提示：首先可以想象谓词p表示有向图的弧，并把各表达式看作图的属性。7.10节中的材料应该能提供一些提示，包括如何找到定义域是某图节点而且谓词p是该图弧的合适模型，或者是如何证明为何一定存在蕴涵。不过请注意，只强调该解释是图，并不足以证明蕴涵。

(3) * 设 S_1 和 S_2 是两个谓词逻辑（或者是命题逻辑，这都没关系）表达式集合，并设它们对应的模型集合分别是 M_1 和 M_2 。

(a) 证明对应表达式集合 $S_1 \cup S_2$ 的模型集合是 $M_1 \cap M_2$ 。

(b) 对应表达式集合 $S_1 \cap S_2$ 的模型集合是否总是 $M_1 \cup M_2$？

(4) * 证明：如果 $(E_1\text{ AND }E_2\text{ AND}\cdots\text{AND }E_n) \rightarrow E$ 是重言式，就有 E_1，E_2，$\cdots$，$E_n \vDash E$ 。

14.11 小结

大家应该从本章中了解到了如下要点。

- 谓词逻辑用原子公式（即带参数的谓词）作为原词操作数，并使用命题逻辑运算符以及两个量词“对所有”和“存在”。
- 谓词逻辑表达式中的变量受量词约束的方式，类似程序中的变量受声明约束的方式。
- 与命题逻辑中的真值赋值不同，在谓词逻辑中我们有一种名为“解释”的更复杂的结构。解释是由定义域、定义域上对应谓词的关系，以及从定义域对应任何自由变量的值组成的。
- 使得表达式集为真的解释就是该表达式集的“模型”。
- 谓词演算的重言式是那些对每种解释而言都能得出TRUE的表达式。尽管很多重言式是通过对命题逻辑重言式进行替换得到的，但也存在一些涉及量词的重要重言式。
- 每个谓词逻辑表达式都可以表示为“前束式”表达式，它是由无量词表达式最后应用量词运算符构成的。

- 谓词逻辑中的证明可以通过与构建命题逻辑中的证明类似的方式构建。
- 在重言式中用常量替换变量可得到另一个重言式，这一推理规则在证明中是很实用的，特别是在处理大量的事实和规则时。
- 如果表达式集 $\{E_1,\cdots,E_n\}$ 的任一模型同时也是表达式E的模型，则该表达式集“蕴涵”表达式E。如果 $E_1,\cdots$，E_n 蕴涵E，在给定 $E_1,\cdots$，E_n 作为前提时就把E视为“真”。
- 哥德尔的定义说明了，如果我们用描述数论（即非负整数的算术）的表达式作为前提，那么对任何证明系统而言，都存在一些由前提蕴涵但不能通过前提证明的表达式。
- 图灵的定理描述了“图灵机”这种正式的计算机模型，并表示存在不能由计算机解决的问题。

14.12　参考文献

12.14节中引用过的介绍逻辑的书籍，包括Enderton [1972]、Mendelson [1987]、Lewis and Papadimitriou [1981]，以及Manna and Waldinger [1990]，也涵盖了有关谓词逻辑的内容。

哥德尔的不完备性定理出现在Gödel [1931]中。图灵有关不可判定性的论文是Turing [1936]。

Gödel, K. [1931]. “Uber formal unentscheidbare satze der Principia Mathematica und verwander systeme,” *Monatschefte fur Mathematik und Physik* **38**, pp. 173–198.

Turing, A. M. [1936]. “On computable numbers with an application to the *entscheidungsproblem*,” *Proc. London Math. Soc.* **2**:42, pp. 230–265.

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks